TECHNISCHE
UNIVERSITÄT
DARMSTADT

LEARNING TO FIND BUGS IN PROGRAMS AND THEIR DOCUMENTATION

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

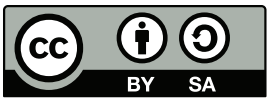
ANDREW HABIB, M.SC.

geboren am 14.06.1989
in Sues, Ägypten

Referenten:

Prof. Dr. Mira Mezini
Prof. Dr. Michael Pradel
Prof. Dr. Premkumar T. Devanbu

Tag der Einreichung: 02.11.2020
Tag der Prüfung: 14.12.2020



ERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

Darmstadt, Deutschland
November 2020

Andrew Habib

ACADEMIC CV

October 2015 - December 2020
Doctoral Degree in Computer Science
Technische Universität Darmstadt, Germany

August 2013 - July 2015
Master of Science Degree in Computer Science and Engineering
Denmark Technical University, Denmark
The Norwegian University of Science and Technology, Norway

September 2006 - June 2011
Bachelor of Science Degree in Computer Science
Bachelor of Science Degree in Mathematics
The American University in Cairo, Egypt

ABSTRACT

Although software is pervasive, almost all programs suffer from bugs and errors. To detect software bugs, developers use various techniques such as static analysis, dynamic analysis, and model checking. However, none of these techniques is bulletproof.

This dissertation argues that *learning from programs and their documentation* provides an effective means to prevent and detect software bugs. The main observation that motivates our work is that *software documentation* is often under-utilized by traditional bug detection techniques. Leveraging the documentation together with the program itself, whether its source code or runtime behavior, enables us to build unconventional bug detectors that benefit from the richness of natural language documentation and the formal algorithm of a program. More concretely, we present techniques that utilize the documentation of a program and the program itself to: (i) Improve the documentation by inferring missing information and detecting inconsistencies, and (ii) Find bugs in the source code or runtime behavior of the program. A key insight we build on is that *machine learning* provides a powerful means to deal with the fuzziness and nuances of natural language in software documentation and that source code is repetitive enough to also allow statistical learning from it. Therefore, several of the techniques proposed in this dissertation employ a learning component whether from documentation, source code, runtime behavior, and their combinations.

We envision the impact of our work to be two-fold. First, we provide developers with novel bug detection techniques that complement traditional ones. Our approaches learn bug detectors end-to-end from data and hence, do not require complex analysis frameworks. Second, we hope that our work will open the door for more research on automatically utilizing natural language in software development. Future work should explore more ideas on how to extract richer information from natural language to automate software engineering tasks, and how to utilize the programs themselves to enhance the state-of-the-practice in software documentation.

ZUSAMMENFASSUNG

Obwohl Software allgegenwärtig ist, leiden fast alle Programme unter Fehlern. Um Softwarefehler zu erkennen, verwenden Entwickler verschiedene Techniken wie statische Analyse, dynamische Analyse und Modellprüfung. Jedoch ist keine dieser Techniken perfekt.

In dieser Dissertation wird argumentiert, dass das Lernen aus Programmen und deren Dokumentation ein wirksames Mittel darstellt, um Softwarefehler zu erkennen und zu verhindern. Die wichtigste Beobachtung, welche diese Arbeit motiviert, ist, dass die Softwaredokumentation von herkömmlichen Fehlererkennungstechniken häufig nicht ausreichend genutzt wird. Durch die Nutzung der Dokumentation zusammen mit dem Programm selbst, unabhängig davon, ob es sich um den Quellcode oder das Laufzeitverhalten handelt, können unkonventionelle Fehlerdetektoren erstellt werden, welche von der Fülle der natürlichen Sprache in der Dokumentation und dem formalen Algorithmus eines Programms profitieren. Konkreter stellen wir Techniken vor, welche die Dokumentation eines Programms und das Programm selbst verwenden, um: (i) die Dokumentation zu verbessern, indem auf fehlende Informationen geschlossen und Inkonsistenzen festgestellt werden, und (ii) Fehler im Quellcode oder im Laufzeitverhalten des Programms zu finden. Eine wichtige Erkenntnis, auf welcher wir aufbauen, ist, dass maschinelles Lernen ein leistungsfähiges Mittel darstellt, um mit der Unschärfe und den Nuancen natürlicher Sprache in der Softwaredokumentation umzugehen, und dass sich der Quellcode oft genug wiederholt, um auch statistisches Lernen daraus zu ermöglichen. Daher verwenden einige der in dieser Dissertation vorgeschlagenen Techniken eine Lernkomponente, welche sich aus Dokumentation, Quellcode, Laufzeitverhalten und deren Kombinationen ergibt.

Wir sehen die Auswirkungen unserer Arbeit in zweifacher Hinsicht. Erstens bieten wir Entwicklern neuartige Fehlererkennungstechniken, welche herkömmliche ergänzen. Unsere Ansätze lernen Fehlerdetektoren durchgängig aus Daten und erfordern daher keine komplexen Analyserahmen. Zweitens hoffen wir, dass unsere Arbeit die Tür für weitere Forschungen zur automatischen Verwendung natürlicher Sprache in der Softwareentwicklung öffnet. Zukünftige Arbeiten sollten weitere Ideen untersuchen, wie umfangreichere Informationen aus der natürlichen Sprache extrahiert

werden können, um Softwareentwicklungsaufgaben zu automatisieren, und wie die Programme selbst verwendet werden können, um die aktuelle Praxis in der Softwaredokumentation zu verbessern.

ACKNOWLEDGEMENTS

Getting a PhD is not an easy journey. However, if you are accompanied by the right people, you would rather enjoy and always cherish it. I was very lucky and blessed to be surrounded by an amazing group of people, without whom my five-year PhD journey would have been an overwhelmingly difficult path to tread.

I would like to start by thanking my PhD adviser, Michael Pradel. One of the most difficult decisions, after deciding to do a PhD, is to choose your adviser (after they choose you of course!). I would always choose to do my PhD with Michael. He has been incredibly supportive, encouraging, and patient. Michael has been a great example and role model, not only as a successful and accomplished researcher, but also as an honest and respectful human being. Michael, I am forever grateful for everything you have taught me and I will always cherish working with you.

Sharing the day-to-day life of a PhD student is an unpleasant experience without kind and nice colleagues. I am therefore grateful to my colleagues and friends from the Software Lab for the warm, fun, and intellectual environment they provided, in order of seniority: Marija Selakovic, Cristian-Alexandru Staicu, Jibesh Patra, and Daniel Lehmann. I will always cherish the time we spent together and I hope we will always stay in touch (thank you Cris for the ML Flamewar group!).

One important person who greatly contributed to my smooth experience in Germany and at the university is our incredible secretary, Andrea Püchner, who at times went out of her way to help beating the German bureaucracy and ensuring that everything works out more than perfectly. I cannot find enough words to thank you, Andrea, for your kind heart and what you have done during my time here.

I would also like to thank Mira Mezini for accepting to be my official local adviser (after Michael moved to the University of Stuttgart) and Prem Devanbu for being an external examiner.

I want to thank Dr. Sherif El-Kassas from my undergraduate studies who instigated my research appetite, gave me a role model to look up to, and has always been supportive.

Next, I would like to thank our friends in Germany who made our life more lively and provided us with warm and sincere company. To George,

Sandra, and Ella; Noufer (now ⲡⲉⲛⲓⲟⲧ ⲩⲟⲧⲣⲏⲥ), Olivia, Gori, and Royce; Lance, Karen, and Adam; Morris and Koki; Amir, Soha, and Alex; and Rano and Nini: Thank you so much for the past five years. To John Gergies: Thank you for picking me from the airport on my first arrival in Germany, and then five years later, for translating my dissertation's abstract to German.

I would like to thank Joseph, Gergis, and Maged, my life-time friends from Egypt for their continuous support, encouragement, and checking on us. I also want to thank Mina Rady and Rania Naguib for visiting us at different times and for their precious company.

To my parents who always believed in me, supported me in every possible way, prayed for me, and encouraged me to go after my dreams: Thank you so much for everything. I will forever be grateful to you! I would also like to thank my elder brother Arsany, his wife Engy, and their kids Abigail and Raphael; and my younger brother Mark for their encouragement and support. Also, I want to thank my parents- and brother-in-law who believed in me and have been encouraging and supportive.

To the love of my life and my partner, Lydia, who has been there for me at my highest and lowest moments, put every effort to encourage and support me, prayed for me, and believed in me: Thank you so much for everything you have done and continue to do for us. I also want to mention that I am super proud of your achievements and accomplishments, which were crowned by your recent master degree in Economics!

I ought to thank our spiritual guides who enlighten our paths and lead us in this life: ⲡⲉⲛⲓⲟⲧ ⲁⲃⲣⲁⲁⲙ for his explanations of God's word, which always give us new understandings and new perspectives, and for his humble spiritual leadership; ⲡⲉⲛⲓⲟⲧ ⲡⲓϫⲱⲗ, whom we already miss after he departed to heaven, for his unconditional love and for giving us a living example of our Lord Jesus Christ; ⲡⲉⲛⲓⲟⲧ ⲩⲟⲧⲣⲏⲥ for showing us how to lead a Godly life without giving up your true and witty self; and Dr. Maher Samuel whom I have never met in person, but I have always been guided by his teachings and philosophy.

Finally, and above all, I want to thank God for all his great deeds in my life. God has always guided my steps, been faithful to me, blessed my ways, and been very generous with me even when I screw up. I know a lot of people, and especially researchers, would take my faith lightly, but I am unable to deny my faith and I cannot not testify about God, his grace, his faithfulness, his righteousness, and his mercy in my life.

CONTENTS

1

INTRODUCTION

1.1 MOTIVATION

Software is a vital component of everyday life nowadays. Mobile devices, personal computers, smart home appliances, transportation systems, medical devices, satellite and communications systems including the Internet, and enterprise solutions are just some examples of how computer programs are ubiquitous. In the EU alone, the software industry is estimated to have a market value of around EUR 229 to EUR 290 billions between 2009 and 2020.[1]

That said, software is unavoidably error-prone. It has been estimated that the number of buggy lines of code in industrially deployed software is about 15–50 per 1,000 lines of code, independent of the underlying programming language [McC04]. Unfortunately, bugs in computer programs can lead to serious system failures. Software failures in 2018 alone were estimated to affect the lives of 3.7 billion people, around half of the world population, and resulted in the loss of USD 1.7 trillions in assets.[2]

Because software bugs are severe and they have drastic impact on individuals' lives as well as the global economy, programmers spend a lot of effort and time to improve the programs they develop trying to eliminate potential errors. To detect programming errors and prevent software failures, developers rely on several techniques:

STATIC ANALYSIS This technique utilizes several abstractions of the program source code, such as abstract interpretation, control-flow, and data-flow analyses to over-approximate the program behavior. After building such approximations into static analysis frameworks, expert

1 https://ec.europa.eu/digital-single-market/en/news/economic-and-social-impact-software-and-services-competitiveness-and-innovation

2 https://www.tricentis.com/resources/software-fail-watch-5th-edition

developers then hand-craft tens or even hundreds of rules for what constitute different kinds of bug patterns. Automatic checkers for these patterns are shipped in many static bug finding tools, such as Google's Error Prone [Aft+12] and Facebook's Infer [Cal+15].

DYNAMIC ANALYSIS This technique observes and analyzes the runtime behavior of the program when executed under different conditions and tries to detect erroneous behaviors and anomalies. Although it detects real faults, crashes, and violations of safety properties, dynamic analysis cannot guarantee the absence of bugs because it relies on under-approximating the program behavior. Dynamic analysis frameworks, such as Jalangi [Sen+13] for JavaScript, and functional testing frameworks, such as Randoop [Pac+07] for Java, are among the most widely used.

MODEL CHECKING This technique formally verifies the program against a specification: A set of desired properties, such as memory safety. Similar to static analysis, model checking does not need to execute the program. Rather, it computes an approximation of the program states and tries to prove the existence (or the absence) of a specific property. Bounded model checkers, such as CBMC [CKL04], exhaustively examine all possible program states up to a bound, but cannot certify the absence of bugs for infinite state programs.

Whether it is static analysis, dynamic analysis, or model checking, no bug finding approach is bullet-proof. These techniques mainly provide best-effort by experts and software developers to reduce the probability of a software bug slipping into production and causing serious failures.

We observe that the main focus of the different techniques for finding and preventing software bugs discussed above is the program itself, i.e., its source code, byte code, or runtime behavior, and sometimes a predefined specification in the case of model checking. However, we notice that other important software artifacts, such as documentation, which are regularly produced by developers as part of the software development process, are neglected by most of the bug finding techniques, although they are more direct, descriptive, and easier to understand. We distinguish between two kinds of software documentation:

USER DOCUMENTATION This kind of documentation is intended for end users of a software system, e.g., user manuals and tutorials.

TECHNICAL DOCUMENTATION This documentation describes the intricacies of a software component in a more technical manner, which is usually intended for other fellow programmers, e.g., API documentation.

Technical software documentation comes in various forms. Some of the most common formats are natural language (NL) description of the semantics of a programming language, such as the Java Language Specification,[3] a structured mix of NL descriptions and names and types (for typed languages) of source code entities, such as Javadoc[4] for Java APIs and JSDoc for JavaScript,[5] and inline source code comments.

On the one hand, these forms of software documentation carry a plethora of valuable information that are often under-utilized by traditional bug finding techniques. On the other hand, such documentation is oftentimes inaccurate, stale, or even worse, presents wrong information about the underlying software [Agh+19], which could lead eventually to serious software failures.

Since current techniques for finding software bugs are far from being perfect, this thesis examines the idea of *utilizing documentation to improve software bug detection techniques.* Our work explores two dimensions: (i) Learning from and leveraging documentation to find bugs in programs, and (ii) Improving documentation by learning from programs source code and runtime behavior.

1.2 TERMINOLOGY

In this work, we use the term *documentation* to refer to technical documentation of software components (vs. user documentation). More specifically, we refer to API documentation just as *documentation*.

Moreover, we use the terms *bug* or *error* interchangeably to refer to either of the following:

- *Programming error:* A bug in the software implementation where the program source code or runtime behavior deviates from the documentation, e.g., the documentation of an API says the method is thread-safe but in practice it results in a dead-lock.

3 https://docs.oracle.com/javase/specs/jls/se14/html/index.html
4 https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html
5 https://jsdoc.app/about-getting-started.html

- *Documentation mistake:* An error in the documentation of an API where a condition or behavior does not match the actual behavior and implementation, e.g., a `NullPointerException` is thrown at runtime when a null argument is passed to the API but the documentation states that `IllegalArgumentException` should be thrown instead.

1.3 LEARNING FROM PROGRAMS AND THEIR DOCUMENTATION

Software documentation is a fundamental artifact in the software engineering process. Developers often communicate their intentions, assumptions, design decisions, algorithmic thoughts, and even usage examples through documentation. That is why documentation is an equivocally important artifact in understanding, correctly using, and maintaining a piece of software. Nevertheless, utilizing documentation in automatic software engineering tasks, e.g., to find bugs, is non-trivial; as well as automatically finding bugs in and improving documentation, for several reasons.

First, developers mostly write documentation in natural language, a fuzzy and vague means of communication [Lak75], making it difficult to leverage such rich resource of information in an automated manner. Second, code bases evolve quickly and not enough resources are available for developers to continuously update the documentation [Agh+20]. Third, documentation best practices are often not clear or even unknown to developers, making it challenging and straining to keep up with and sustain good documentation.

To tackle the fuzziness of the developers' intentions, and the vagueness and nuances of natural language, we propose utilizing probabilistic techniques to learn from documentation to improve source code. To cope up with fast evolving and growing source code, we also leverage learning, from source code (or its runtime behavior), to capture the semantics and actual behavior of the program and reflect them on the documentation. In both directions, our decision to use machine learning is rooted in the fact that probabilistic learning has matured enough to provide powerful algorithms for such tasks as well as the availability of large open-source code bases to learn from.

Therefore, this dissertation argues that:

> *Automatically learning from programs and their documentation provides an effective means to prevent and detect software bugs.*

1.4 CONTENTS AND CONTRIBUTIONS

In this dissertation, we present an empirical study and four approaches to support our thesis statement that leveraging programs together with their documentation can improve the tooling and support for automatic software bug detection and prevention. First, we present a novel study that reports on state-of-the-art techniques in static bug finding and identify several of their limitations. Then, we contribute three effective approaches that showcase how to automatically leverage programs and their documentation to detect software bugs and improve the documentation. Finally, we present a futuristic, and possibly an alternative, approach to traditional static bug detection. Figure 1.1 shows a high-level summary of the contribution of each chapter and which software artifacts each approach or study utilizes. In the following, we summarize each contribution highlighting key ideas and findings.

CHAPTER 2: THE STATE OF STATIC BUG DETECTORS

In this chapter, we present an empirical study of the recall of three state-of-the-art static bug finding tools, namely, Google's Error Prone [Aft+12], Facebook's Infer [Cal+15], and SpotBugs [HP04], the successor of FindBugs. Usually, the evaluation of static bug finding techniques focuses on their precision, i.e., how often do they report false positives. However, we show that these mature and industry deployed tools suffer from low recall too, i.e., they miss many of the bugs in a large set of real-world bugs.

This study highlights the need for improving traditional static bug finders and identifies several classes of bug patterns which are missed by static bug finders. One of the observations of this study is that domain-specific knowledge is crucial for detecting a class of bugs missed by the traditional tools. We argue that such knowledge could be, in part, found in software documentation.

CHAPTER 3: INFERRING THREAD SAFETY DOCUMENTATION

In this chapter, we address the problem of undocumented thread safety behavior of classes in object-oriented languages. First, we present an empirical study of how many Java classes are documented to be thread-safe or not. The findings of the study are consistent with several blog posts and issue reports by developers where we find that thread safety is often an under-documented property in Java classes. We then propose a novel

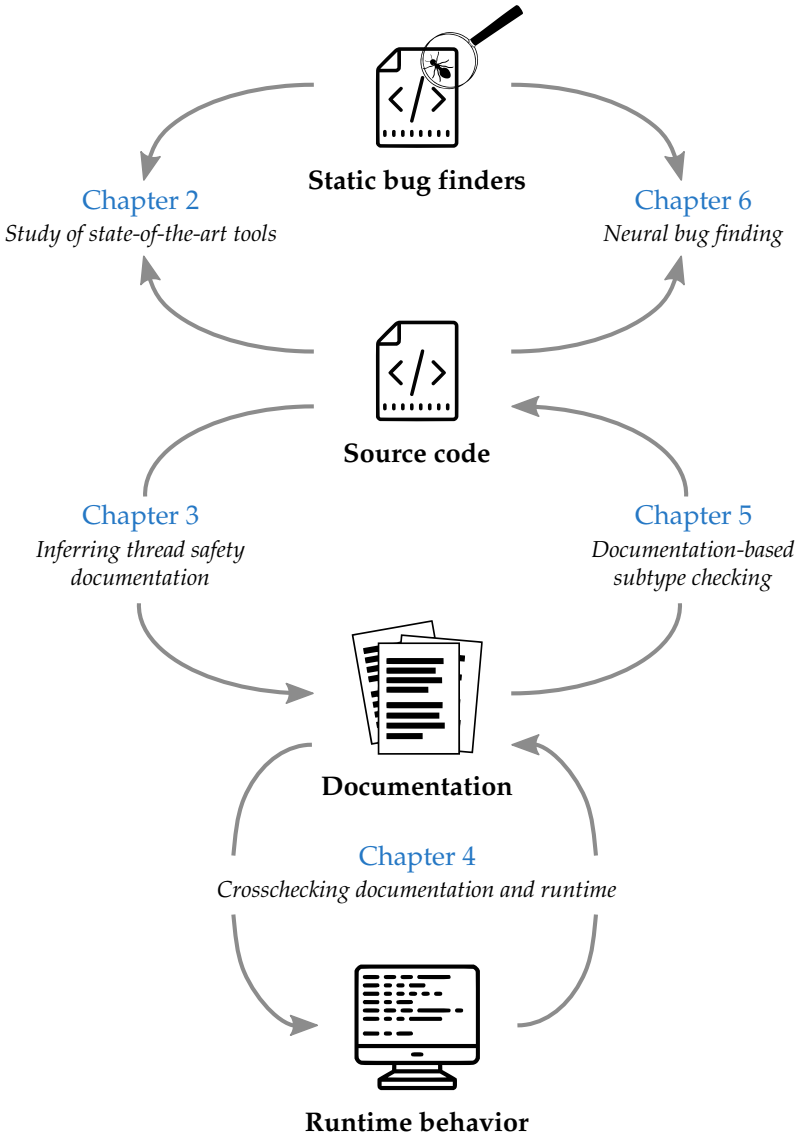


FIGURE 1.1: Overview of the contributions and techniques proposed in each chapter of the dissertation and the different software artifacts each chapter builds on.

machine learning technique which learns from an API source code to infer whether a class is thread-safe or not.

This approach leverages a light-weight static analysis combined with a learning-based approach to improve APIs documentation regarding a critical property: multi-threading. We show that our approach is effective and efficient and could be easily used to improve the documentation of libraries to prevent concurrency-related bugs.

Chapter 4: crosschecking documentation and runtime

In this chapter, we introduce a learning-based approach to crosscheck an API documentation against its observed exceptional runtime behavior. The intuition of this technique is that documentation should describe the true behavior of an API. Discrepancies between an API documentation and its observed runtime behavior mean that either the API implementation is itself buggy, or its documentation is not correct.

This technique serves as an automated oracle for the runtime behavior of programs. Indeed, some of the bugs it reports were fixed by changing the API code, and others were fixed by updating the documentation.

Chapter 5: from documentation to static subtype checking

In this chapter, we leverage a ubiquitous form of API documentation, JSON schemas, to provide static subtype checking for data-intensive applications. JSON schemas are widely used to describe REST APIs, store and retrieve data in NoSQL databases, and in machine learning applications. We propose an algorithm that decides in a reasonable time, for a large set of the JSON Schema language, whether two schemas are subtype of each other.

We implemented our algorithm in an open-source tool that is deployed as part of an automated machine learning library, developed and used by IBM AI, to detect data compatibility bugs in machine learning pipelines. Moreover, we show that this algorithm detects API evolution errors in other domains, such as cloud computing.

Chapter 6: neural bug finding: a futuristic outlook

In this chapter, we investigate a novel and potentially alternative approach to traditional bug finding techniques, neural bug finding. This work examines the idea of leveraging a neural model to learn how to detect instances of frequent bug patterns. We propose a simple approach that exploits two dimensions of the bi-modality of source code: (i) The natural

language aspect communicated through the identifiers and types names, and (ii) The algorithmic channel expressed in the programming language syntax and keywords; both available in the sequence of tokens of the program source code. Although the proposed technique does not come on par with state-of-the-art bug finding approaches yet, it serves as a stepping stone towards non-traditional techniques which supplement classical bug detection tools.

Finally, we provide an overview of the related work in Chapter 7 and conclude the dissertation in Chapter 8 by highlighting future research directions in learning from and leveraging programs and their documentation in software engineering tasks.

1.5 LIST OF PUBLICATIONS AND OPEN-SOURCE IMPLEMENTATION

Parts of the work in this dissertation are based on the following peer-reviewed publications and technical reports from which it verbatim reuses material:

1. [HP18a] Andrew Habib and Michael Pradel. *How many of all bugs do we find? a study of static bug detectors.* IEEE/ACM International Conference on Automated Software Engineering (ASE) 2018.

2. [HP18b] Andrew Habib and Michael Pradel. *Is this class thread-safe? inferring documentation using graph-based learning.* IEEE/ACM International Conference on Automated Software Engineering (ASE) 2018.

3. [Hab+19] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. *Type Safety with JSON Subschema.* CoRR abs/1911.12651 (2019).

4. [HP19] Andrew Habib and Michael Pradel. *Neural Bug Finding: A Study of Opportunities and Challenges.* CoRR abs/1906.00307 (2019).

Table 1.1 shows the mapping between the publications above and the chapters in this dissertation.

Additionally, to facilitate the reproducibility of our results and future research, we make available several of the artifacts, implementations, and datasets we produced as part of this dissertation. Table 1.2 lists the different chapters and their corresponding artifacts.

TABLE 1.1: Mapping of publications to chapters in this dissertation.

[HP18a]	Chapter 2
[HP18b]	Chapter 3
[Hab+19]	Chapter 5
[HP19]	Chapter 6

TABLE 1.2: Dissertation chapters and their corresponding public artifacts.

	Artifact link	Contents
Chapter 2	`https://github.com/sola-da/StaticBugCheckers`	Detailed results and source code
Chapter 3	`https://github.com/sola-da/TSFinder`	Tool incl. source code and dataset
Chapter 5	`https://github.com/IBM/jsonsubschema`	Tool incl. source code and pypi module

2

THE STATE OF STATIC BUG DETECTORS

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: *How many of all real-world bugs do static bug detectors find?* This chapter addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs.

2.1 MOTIVATION

Finding software bugs is an important but difficult task. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [Cho+01; Pal+11]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic loses and even killed people [Lio96; Pou04; ZC09].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs, e.g., collect information about abnormal runtime behavior, crashes, and violations of safety properties, e.g., expressed through

assertions. Finally, developers use static bug detection tools, which check the source code or parts of it for potential bugs.

In this chapter, we focus on static bug detectors because they have become increasingly popular in recent years and are now widely used by major software companies. Popular and widely used tools include Google's Error Prone [Aft+12], Facebook's Infer [Cal+15], or SpotBugs, the successor to the widely used FindBugs tool [HP04]. These tools are typically designed as an analysis framework based on some form of static analysis that scales to complex programs, e.g., AST-based pattern matching or data-flow analysis. Based on the framework, the tools contain an extensible set of checkers that each addresses a specific bug pattern, i.e., a class of bugs that occurs across different code bases. Typically, a bug detector ships with dozens or even hundreds of patterns. The main benefit of static bug detectors compared to other bug finding approaches is that they find bugs early in the development process, possibly right after a developer introduces a bug. Furthermore, applying static bug detectors does not impose any special requirements, such as the availability of tests, and can be fully automated.

The popularity of static bug detectors and the growing set of bug patterns covered by them raise a question: *How many of all real-world bugs do these bug detectors find?* Or in other words, what is the recall of static bug detectors? Answering this question is important for several reasons. First, it is an important part of assessing the current state-of-the-art in automatic bug finding. Most reported evaluations of bug finding techniques focus on showing that a technique detects bugs and how precise it is, i.e., how many of all reported warnings correspond to actual bugs rather than false positives. We do not consider these questions here. In contrast, practically no evaluation considers the above recall question. The reason for this omission is that the set of "all bugs" is unknown (otherwise, the bug detection problem would have been solved), making it practically impossible to completely answer the question. Second, understanding the strengths and weaknesses of existing static bug detectors will guide future work toward relevant challenges. For example, better understanding of which bugs are currently missed may enable future techniques to cover previously ignored classes of bugs. Third, studying the above question for multiple bug detectors allows us to compare the effectiveness of existing tools with each other: Are existing tools complementary to each other or does one tool subsume another one? Fourth and finally, studying the above question will provide an estimate of how close the current state-of-the-art is to the ultimate, but admittedly unrealistic, goal of finding all bugs.

2.2 METHODOLOGY

This section presents our methodology for studying which bugs are detected by static bug detectors. At first, we describe the bugs (Section 2.2.1) and bug detection tools (Section 2.2.2) that we study. Then, we present our experimental procedure for identifying and validating matches between the warnings reported by the bug detectors and the real-world bugs (Section 2.2.3). Finally, we discuss threats to validity in Section 2.5.

2.2.1 *Real-World Bugs*

Our study builds on an extended version of the Defects4J data set, a collection of bugs from popular Java projects. In total, the data set consists of 597 bugs that are gathered from different versions of 15 projects. We use Defects4J for this study for three reasons. First, it provides a representative set of real-world bugs that has been gathered independently of our work. The bugs cover a wide spectrum of application domains and have been sampled in a way that does not bias the data set in any relevant way. Second, the data set is widely used for other bug-related studies, e.g., on test generation [Sha+15], mutation testing [Jus+14], fault localization [Pea+17], and bug repair [Mar+17], showing that is has been accepted as a representative set of bugs. Third, Defects4J provides not only bugs but also the corresponding bug fixes, as applied by the actual developers. Each bug is associated with two versions of the project that contains the bug: a buggy version, just before applying the bug fix, and a fixed version, just after applying the bug fix. The bug fixes have been isolated by removing any irrelevant code changes, such as new features or refactorings. As a result, each bug is associated with one or more Java classes, i.e., source code files that have been modified to fix the bug. The availability of fixes is important not only to validate that the developers considered a bug as relevant, but also to understand its root cause.

Defects4J is continuously growing with 10 officially released versions so far.[1] At the time of this study, the official version of Defects4J (version 1.1.0) consisted of 395 bugs collected from 6 Java projects. An unofficial pull request extends the dataset with 202 additional bugs from 9 additional projects.[2,3] In our work, we use the extended version of the dataset and refer

1 https://github.com/rjust/defects4j
2 https://github.com/rjust/defects4j/pull/112
3 The pull request was merged in a later release.

TABLE 2.1: Projects and bugs of Defects4J.

Project ID	Project Name	Bugs	
	Official Defects4J		
Chart	JFreeChart	26	
Closure	Google Closure	133	
Lang	Apache commons-lang	64	(65)
Math	Apache commons-math	106	
Mockito	Mockito framwork	38	
Time	Joda-Time	27	
6 projects from official release		394	(395)
	Extended Defects4J		
Codec	Apache commons-codec	21	(22)
Cli	Apache commons-cli	24	
Csv	Apache commons-csv	12	
JXPath	Apache commons-JXPath	14	
Guava	Guava library	9	
JCore	Jackson core module	13	
JDatabind	Jackson data binding	39	
JXml	Jackson XML utilities	5	
Jsoup	Jsoup HTML parser	63	(64)
9 projects from pull request		200	(202)
Total of **15** projects		**594**	(597)

to it as "Defects4J". Table 2.1 lists the projects and bugs in the data set.[4] We exclude three of the 597 bugs for technical reasons: Lang-48 because Error Prone does not support Java 1.3, and Codec-5 and Jsoup-4 because they introduce a new class in the bug fix, which does not match our methodology that relies on analyzing changes to existing files.

4 We refer to bugs using the notation ProjectID-N, where N is a unique number.

2.2.2 *Static Bug Detectors*

We study three static bug detectors for Java: (i) Error Prone [Aft+12], a tool developed by Google and is integrated into their Tricorder static analysis ecosystem [Sad+15]; (ii) Infer [Cal+15], a tool developed and used internally by Facebook; and (iii) SpotBugs, the successor of the pioneering FindBugs [HP04] tool. These tools are used by professional software developers. For example, Error Prone and Infer are automatically applied to code changes to support manual code review at Google and Facebook, respectively. All three tools are available as open-source. We use the tools with their default configuration.

GOOGLE'S ERROR PRONE Error Prone is an open-source static analysis tool written in Java. It is developed, maintained, and used internally by Google to detect common errors in Java code. The tool is designed as a Java compiler hook and can easily integrate with most common build systems, or even could be hooked to the Java compiler without any build tool. Error Prone can suggest code fixes to its reported bugs and can automatically patch the erroneous code. It requires at least JDK 8 to run and can compile Java 6 and 7 source code. Error Prone defines a set of rules corresponding to known Java bug patterns and categorizes its findings into warnings and errors. Moreover, the tool provides a set of experimental checks that are disabled by default. In our experiments, we use Error Prone with its default checkers.

FACEBOOK'S INFER Infer is a static analysis tool developed by Facebook, which was open-sourced in 2015. It is written in OCaml and analyzes Java, C, C++, and Objective-C source code for a variety of issues. Similar to Error Prone, the tool can be used in combination with many Java build systems or as a stand alone tool in combination with a Java compiler. Infer runs in two phases, the first is called the translation phase, where information from the Java compiler is gathered during a normal compilation session, while Infer performs its own translation of Java code into an intermediate format. The second phase is the analysis, where several of Infer internal checkers are applied to the intermediate files captured during the translation phase. Infer builds on formal program analysis techniques, such as abstract interpretation and bi-abduction logic. Similar to Error Prone, Infer has two sets of checkers, a default set and an experimental set. For our experiments, we use Infer with its default settings.

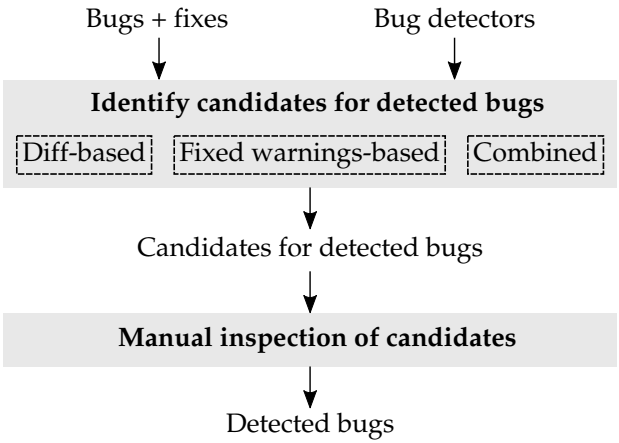


FIGURE 2.1: Overview of our methodology.

SPOTBUGS SpotBugs is the successor of the pioneering FindBugs Java tool. SpotBugs is written in Java and can run as a standalone static analysis tool to analyze Java projects and files or it could be integrated within many Java build tools. The tool reports a variety of warnings that fall into different categories, such as correctness, bad-practice, and malicious code with a severity score associated with each warning. Similar to Error Prone and Infer, we use SpotBugs with its default configuration.

SpotBugs is quite similar to Error Prone as it uses predefined rules to discover known bug patterns. Even the Error Prone documentation page mentions that "Eclipse users should use the Findbugs Eclipse plugin instead, as it catches many of the same issues." However, Error Prone being actively used at Google, it undergoes continuous improvements and tuning, where its developers report less than 10% false positive rate [Sad+15].

2.2.3 *Experimental Procedure*

Given a set of bugs and a set of static bug detectors, the overall goal of the methodology is to identify those bugs among the set B of provided bugs that are detected by the given tools. We represent a *detected bug* as a tuple (b, w), where $b \in B$ is a bug and w is a warning that points to the buggy code. A single bug b may be detected by multiple warnings, e.g., (b, w_1) and (b, w_2), and a single warning may point to multiple bugs, e.g., (b_1, w) and (b_2, w).

A naive approach to assess whether a tool finds a particular bug would be to apply the tool to the buggy version of the code and to manually inspect each reported warning. Unfortunately, static bug detectors may produce many warnings and manually inspecting each warning for each buggy version of a program does not scale to the number of bugs we consider. Another possible approach is to fully automatically match warnings and bugs, e.g., by assuming that every warning at a line involved in a bug fix points to the respective bug. While this approach solves the scalability problem, it risks to overapproximate the number of detected bugs. The reason is that some warnings may coincidentally match a code location involved in a bug, but nevertheless do not point to the actual bug.

Our approach to identify detected bugs is a combination of automatic and manual analysis, which reduces the manual effort compared to inspecting all warnings while avoiding the overapproximation problem of a fully automatic matching. To identify the detected bugs, we proceed in two main steps, as summarized in Figure 2.1. The first step automatically identifies candidates for detected bugs, i.e., pairs of bugs and warnings that are likely to match each other. We apply three variants of the methodology that differ in how to identify such candidates:

- an approach based on differences between the code before and after fixing the bug,
- an approach based on warnings reported for the code before and after fixing the bug, and
- the combination of the two previous approaches.

The second step is to manually inspect all candidates to decide which bugs are indeed found by the bug detectors. This step is important to avoid counting coincidental matches as detected bugs.

2.2.3.1 *Identifying Candidates for Detected Bugs*

COMMON DEFINITIONS We explain some terms and assumptions used throughout this section. Given a bug b, we are interested in the set L_b of *changed lines*, i.e., lines that were changed when fixing the bug. We assume that these lines are the locations where developers expect a static bug detector to report a warning. In principle, this assumption may not hold because the bug location and the fix location may differ. We further discuss this potential threat to validity in Section 2.5. We compute the changed lines based on the differences, or short, the diff, between the code just

before and just after applying the bug fix. The diff may involve multiple source code files. We compute the changed lines as lines that are modified or deleted, as these are supposed to directly correspond to the bug. In addition, we consider a configurable window of lines around the location of newly added lines. As a default value, we use a window size of [-1,1].

Applying a bug detector to a program yields a set of warnings. We refer to the sets of warnings for the program just before and just after fixing a bug b as $W_{before}(b)$ and $W_{after}(b)$, or simply W_{before} and W_{after} if the bug is clear from the context. The bug detectors we use can analyze entire Java projects. Since the purpose of our study is to determine whether specific bugs are found, we apply the analyzers only to the files involved in the bug fix, i.e., files that contain at least one changed line $l \in L_b$. We also provide each bug detector the full compile path along with all third-party dependencies of each buggy or fixed program so that inter-project and third-party dependencies are resolved. The warnings reported when applying a bug detector to a file are typically associated with specific line numbers. We refer to the lines that are flagged by a warning w as $lines(w)$.

DIFF-BASED METHODOLOGY One approach to compute a set of candidates for detected bugs is to rely on the diff between the buggy and the fixed versions of the program. The intuition is that a relevant warning should pinpoint one of the lines changed to fix the bug. In this approach, we perform the following for each bug and bug detector:

1. Compute the lines that are flagged with at least one warning in the code just before the bug fix:

$$L_{warnings} = \bigcup_{w \in W_{before}} lines(w)$$

2. Compute the candidates of detected bugs as all pairs of a bug and a warning where the changed lines of the bug overlap with the lines that have a warning:

$$B^{diff}_{cand} = \{(b, w) \mid L_b \cap L_{warnings} \neq \emptyset\}$$

For example, the bug in Figure 2.2a is a candidate for a bug detected by SpotBugs because the tool flagged line 55, which is also in the set of changed lines.

FIXED WARNINGS-BASED METHODOLOGY As an alternative approach for identifying a set of candidates for detected bugs, we compare the warnings reported for the code just before and just after fixing a bug. The intuition is that a warning caused by a specific bug should disappear when fixing the bug. In this approach, we perform the following for each bug and bug detector:

1. Compute the set of fixed warnings, i.e., warnings that disappear after applying the bug fix:

$$W_{fixed} = W_{before} \setminus W_{after}$$

2. Compute the candidates for detected bugs as all pairs of a bug and a warning where the warning belongs to the fixed warnings set:

$$B_{cand}^{fixed} = \{(b, w) \mid w \in W_{fixed}\}$$

In this step, we do not match warning messages based on line numbers because line numbers may not match across the buggy and fixed files due to added and deleted code. Instead, we compare the messages based on the warning type, category, severity, rank, and code entity, e.g., class, method, and field.

For example, Figure 2.2c shows a bug that the fixed warnings-based approach finds as a candidate for a detected bug by Error Prone because the warning message reported at line 175 disappears in the fixed version. In contrast, the approach misses the candidate bug in Figure 2.2a because the developer re-introduced the same kind of bug in line 62, and hence, the same warning is reported in the fixed code.

COMBINED METHODOLOGY Both the diff-based and the fixed warnings-based approaches may yield different candidates for detected bugs. For instance, both approaches identify the bugs in Figure 2.2c and Figure 2.2d as candidates, whereas only the diff-based approach identifies the bugs in Figure 2.2a and Figure 2.2b. Therefore, we consider as a third variant of our methodology: *the combination* of the fixed warnings- and the diff-based approaches:

$$B_{cand}^{combine} = B_{cand}^{diff} \cup B_{cand}^{fixed}$$

Unless otherwise mentioned, the combined methodology is the default in the remainder of the chapter.

2.2.3.2 *Manual Inspection and Classification of Candidates*

The automatically identified candidates for detected bugs may contain coincidental matches of a bug and warning. For example, suppose that a bug detector warns about a potential null dreference at a specific line and that this line gets modified as part of a bug fix. If the fixed bug is completely unrelated to dereferncing a null object, then the warning would not have helped a developer in spotting the bug.

To remove such coincidental matches, we manually inspect all candidates for detected bugs and compare the warning messages against the buggy and fixed versions of the code. We classify each candidate into one of three categories: (i) If the warning matches the fixed bug and the fix modifies lines that affect the flagged bug only, then this is a *full match*. (ii) If the fix targets the warning but also changes other lines of code not relevant to the warning, then it is a *partial match*. (iii) If the fix does not relate to the warning message at all, then it is a *mismatch*.

For example, the bug in Figure 2.2d is classified as a full match since the bug fix exactly matches the warning message: to prevent a `NullPointerException` on the value returned by `ownerDocument()`, a check for nullness has been added in the helper method `getOutputSettings()`, which creates an empty `Document("")` object when `ownerDocument()` returns null.

As an example of a partial match, consider the bug in Figure 2.2a. As we discussed earlier in Section 2.2.3.1, the developer attempted a fix by applying proper check and cast in lines 58-63 of the fixed version. We consider this candidate bug a partial match because the fixed version also modifies line 60 in the buggy file by changing the return value of the method `hashCode()`. This change is not related to the warning reported by SpotBugs. It is worth noting that the fact that the developer unfortunately re-introduced the same bug in line 62 of the fixed version does not contribute to the partial matching decision.

Finally, the bug in Figure 2.2b is an example of a mismatch because the warning reported by Error Prone is not related to the bug fix.

Buggy code:

```
53 @Override
54 public boolean equals(Object o) {
55   return method.equals(o);
56 }
57
58 @Override
59 public int hashCode() {
60   return 1;
61 }
```

Fixed code:

```
53 @Override
54 public boolean equals(Object o) {
55   if (this == o) {
56     return true;
57   }
58   if (o instanceof DelegatingMethod) {
59     DelegatingMethod that = (DelegatingMethod) o;
60     return method.equals(that.method);
61   } else {
62     return method.equals(o);
63   }
64 }
65
66 @Override
67 public int hashCode() {
68   return method.hashCode();
69 }
```

(a) Bug Mockito-11. Warning by SpotBugs at line 55: Equality check for operand not compatible with `this`. L_b = { 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 67, 68, 69 }. Found by diff-based methodology. Classification: Partial match.

FIGURE 2.2: Candidates for detected bugs and their manual classification.

(Continued on next page)

Buggy code:

```
1602 public Dfp multiply(final int x) {
1603   return multiplyFast(x);
1604 }
```

Fixed code:

```
1602 public Dfp multiply(final int x) {
1603   if (x >= 0 && x < RADIX) {
1604     return multiplyFast(x);
1605   } else {
1606     return multiply(newInstance(x));
1607   }
1608 }
```

(b) Bug Math-17. Warning by Error Prone at line 1602: Missing `@Override`. L_b = { 1602, 1603, 1604, 1605, 1606, 1607, 1608 }. Found by diff-based methodology. Classification: Mismatch.

Buggy code:

```
173 public Week(Date time, TimeZone zone) {
174   // defer argument checking...
175   this(time, RegularTimePeriod.DEFAULT_TIME_ZONE,
          Locale.getDefault());
176 }
```

Fixed code:

```
173 public Week(Date time, TimeZone zone) {
174   // defer argument checking...
175   this(time, zone, Locale.getDefault());
176 }
```

(c) Bug Chart-8. Warning by Error Prone at line 175: Chaining constructor ignores parameter. L_b = { 175 }. Found by: Diff-based methodology and fixed warnings-based methodology. Classification: Full match.

FIGURE 2.2: Candidates for detected bugs and their manual classification.

(Continued on next page)

Buggy code:

```
214 public Document ownerDocument() {
215   if (this instanceof Document)
216     return (Document) this;
217   else if (parentNode == null)
218     return null;
219   else
220     return parentNode.ownerDocument();
221 }
    :
362 protected void outerHtml(StringBuilder accum) {
363   new NodeTraversor(new OuterHtmlVisitor(accum,
          ownerDocument().outputSettings())).traverse(this);
364 }
```

Fixed code:

```
362 protected void outerHtml(StringBuilder accum) {
363   new NodeTraversor(new OuterHtmlVisitor(accum,
          getOutputSettings())).traverse(this);
364 }
365
366 // if this node has no document (or parent), retrieve the
        default output settings
367 private Document.OutputSettings getOutputSettings() {
368   return ownerDocument() != null ?
          ownerDocument().outputSettings() :
          (new Document("")).outputSettings();
369 }
```

(d) Bug Jsoup-59. Warning by Infer at line 363: `null` dereference. L_b = { 363, 364, 365, 366, 367, 368, 369 }. Found by: Diff-based methodology and fixed warnings-based methodology. Classification: Full match.

FIGURE 2.2: Candidates for detected bugs and their manual classification.

2.2.3.3 *Error Rate*

Beyond the question of how many of all bugs are detected, we also consider the *error rate* of a bug detector. Intuitively, it indicates how many warnings the bug detector reports. We compute the error rate by normalizing the number of reported warnings to the number of analyzed lines of code:

$$ER = \frac{\sum_{b \in B} |W_{before}(b)|}{\sum_{b \in B} \sum_{f \in files(b)} LoC(f)}$$

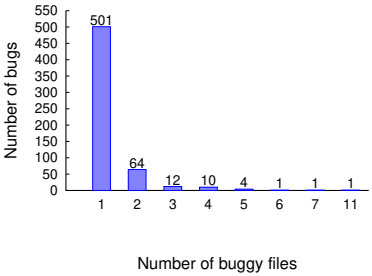
where $files(b)$ are the files involved in fixing bug b and $LoC(f)$ yields the number of lines of code of a file.

2.3 IMPLEMENTATION

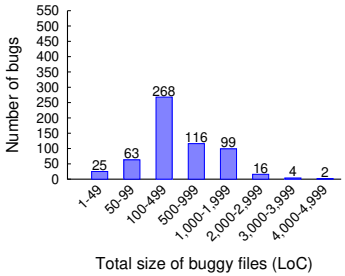
To apply the bug detectors to the files relevant for a particular bug, we apply them to single Java files (Error Prone and Infer) or to a list of classes (SpotBugs). All tools we use, Error Prone, Infer, and SpotBugs, accept as parameters the full compile path of a project along with the path to the desired source file (Error Prone and Infer) or qualified class name (SpotBugs) to check for errors. Each tool reports a set of warnings or errors at different source code locations along with details about each warning and short message to explain the flagged problem. We represent the warnings from different tools in a single data format, as JSON files, based on which we perform the automatic search for candidates of detected bugs.

2.4 EXPERIMENTAL RESULTS

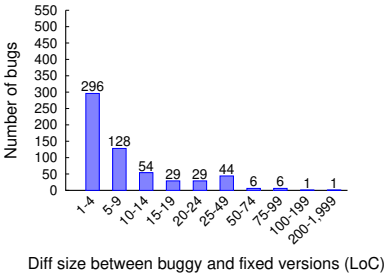
This section presents the results of applying our methodology to 594 bugs and three bug detectors. We start by describing some properties of the studied bugs (Section 2.4.1) and the warnings reported by the bug detectors (Section 2.4.2). Next, we report on the candidates for detected bugs (Section 2.4.3) and how many of them could be manually validated and their kinds (Section 2.4.4), followed by a comparison of the studied bug detectors (Section 2.4.5). To better understand the weaknesses of current bug detectors, Section 2.4.6 discusses why the detectors miss many bugs. Finally, Section 2.4.7 empirically compares the three variants of our methodology.



(a) Number of buggy files.



(b) Bugs by total size of buggy files.



(c) Total size of diffs between buggy and fixed files.

FIGURE 2.3: Properties of the studied bugs.

2.4.1 *Properties of the Studied Bugs*

To better understand the setup of our study, we measure several properties of the 594 studied bugs. Figure 2.3a shows how many files are involved in fixing a bug. For around 85% of the bugs, the fix involves changing a single source code file. Figure 2.3c shows the number of lines of code in the diff between the buggy and the fixed versions. This measure gives an idea of how complex the bugs and their fixes are. The results show that most bugs involve a small number of lines: For 424 bugs, the diff size is between one and nine lines of code. Two bugs have been fixed by modifying, deleting, or inserting more than 100 lines.

TABLE 2.2: Warnings generated by each tool. The minimum, maximum, and average numbers of warnings are per bug in the Defects4J and consider all files involved in the bug fix.

	Warnings				
	Per bug				
Tool	Min	Max	Avg	Total	Error rate
Error Prone	0	148	7.58	4,402	0.01225
Infer	0	36	0.33	198	0.00055
SpotBugs	0	47	1.1	647	0.0018
Total				**5,247**	

2.4.2 *Warnings Reported by the Bug Detectors*

The first step in our methodology is running each tool on all files involved in each of the bugs. Table 2.2 shows the minimum, maximum, and average number of warnings per bug, i.e., in the files involved in fixing the bug, the total number of warnings reported by each tool, and the error rate as defined in Section 2.2.3.3. We find that Error Prone reports the highest number of warnings, with a maximum of 148 warnings and an average of 7.58 warnings per bug. This is also reflected by an error rate of 0.01225.

The studied bug detectors label each warning with a description of the potential bug. Table 2.3 shows the top 5 kinds of warnings reported by each tool. The most frequent kind of warning by Error Prone is about missing the `@Override` annotation when a method overrides a method with the same signature in its parent class. Infer's most reported kind of warning complains about a potential null dereference. Finally, the most frequent kind of warning by SpotBugs is related to missing the `default` case in a `switch` statement. The question how many of these warnings point to a valid problem (i.e., true positives) is outside of the scope of this study.

2.4.3 *Candidates for Detected Bugs*

Given the number of reported warnings, which totals to 5,247 (Table 2.2), it would be very time-consuming to manually inspect each warning. The automated filtering of candidates for detected bugs yields a total of 153

TABLE 2.3: Top 5 warnings reported by each static checker.

Warning	Count
Error Prone	
Missing `@Override`	3211
Comparison using reference equality	398
Boxed primitive constructor	234
Operator precedence	164
Type parameter unused in formals	64
Infer	
`null` dereference	90
Thread safety violation	43
Unsafe `@GuardedBy` access	30
Resource leak	29
Method with mutable return type returns immutable collection	1
SpotBugs	
`switch` without `default`	109
Inefficient `Number` constructor	79
Read of unwritten field	45
Method naming convention	37
Reference to mutable object	31

warnings and 89 candidates (Table 2.4), which significantly reduces the number of warnings and bugs to inspect. Compared to all reported warnings, the selection of candidates reduces the number of warnings by 97%.

The number of warnings is greater than the number of candidates because we count warnings and candidates obtained from all tools together and each tool could produce multiple warnings per line(s).

2.4.4 *Validated Detected Bugs*

To validate the candidates for detected bugs, we inspect each of them manually. Based on the inspection, we classify each candidate as a full match, a partial match, or a mismatch, as described in Section 2.2.3.2. Overall, the three tools found 31 bugs, as detailed in the table in Figure 2.4. After removing duplicates, i.e., bugs found by more than one tool, there are 27 unique validated detected bugs.

We draw two conclusions from these results. First, the fact that 27 unique bugs are detected by the three studied bug detectors shows that these tools would have had a non-negligible impact, if they would have been used during the development of the studied programs. This result is encouraging for future work on static bug detectors and explains why several static bug detection tools have been adopted in industry. Second, even when counting both partial and full matches, the overall bug detection rate of all three bug detectors together is only 4.5%. While reaching a detection anywhere close to 100% is certainly unrealistic, e.g., because some bugs require a deep understanding of the specific application domain, we believe that the current state-of-the-art leaves room for improvement.

To get an idea of the kinds of bugs the checkers find, we describe the most common patterns that contribute to finding bugs. Out of the eight bugs found by Error Prone, three are due to missing an `@Override` annotation, and two bugs because the execution may fall through a `switch` statement. For the five bugs found by Infer, four bugs are potential `null` deferences. Out of the 18 bugs detected by SpotBugs, three are discovered by pointing to dead local stores (i.e., unnecessarily computed values), and two bugs are potential `null` deferences. Finally, the two bugs found by both Infer and SpotBugs are `null` deferences, whereas the two bugs found by both Error Prone and SpotBugs are a string format error and an execution that may fall through a `switch` statement.

2.4.5 *Comparison of Bug Detectors*

The right-hand side of Figure 2.4 shows to what extent the bug detectors complement each other. SpotBugs finds most of the bugs, 18 of all 27, of which 14 are found only by SpotBugs. Error Prone finds 6 bugs that are not found by any other tool, and Infer finds 3 bugs missed by the other tools. We conclude that the studied tools complement each other to a large extent, suggesting that developers may want to combine multiple tools and

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
Total	**31**
Total of **27 unique bugs**	

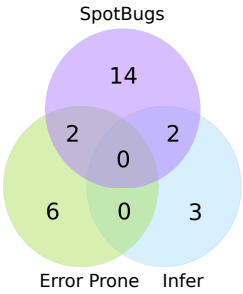


FIGURE 2.4: Total number of bugs found by all three static checkers and their overlap.

that researchers could address the problem of how to reconcile warnings reported by different tools.

2.4.6 *Reasons for Missed Bugs*

To better understand why the vast majority of bugs are not detected by the studied bug detectors, we manually inspect and categorize some of the missed bugs. We inspect a random sample of 20 of all bugs that are not detected by any bug detector. For each sampled bug, we try to understand the root cause of the problem by inspecting the diff and by searching for any issue reports associated with the bug. Next, we carefully search the list of bug patterns supported by the bug detectors to determine whether any of the detectors could have matched the bug. If there is a bug detector that relates to the bug, e.g., by addressing a similar bug pattern, then we experiment with variants of the buggy code to understand why the detector has not triggered an alarm. Based on this process, we have the following two interesting findings.

DOMAIN-SPECIFIC BUGS First, the majority of the missed bugs (14 out of 20) are domain-specific problems not related to any of the patterns supported by the bug checkers. The root causes of these bugs are mistakes in the implementation of application-specific algorithms, typically because the developer forgot to handle a specific case. Moreover, these bugs manifest in ways that are difficult to identify as unintended without domain knowledge, e.g., by causing an incorrect string to be printed or an incorrect number to be computed. For example, Math-67 is a bug in the implementation

of a mathematical optimization algorithm that returns the last computed candidate value instead of the best value found so far. Another example is Closure-110, a bug in a JavaScript compiler that fails to properly handle some kinds of function declarations. Finally, Time-14 is due to code that handles dates but forgot to consider leap years and the consequences of February 29.

NEAR MISSES Second, some of the bugs (6 out of 20) could be detected with a more powerful variant of an existing bug detector. We distinguish two subcategories of these bugs. On the one hand, the root causes of some bugs are problems targeted by at least one existing bug detector, but the current implementation of the detector misses the bug. These bugs manifest through a behavior that is typically considered unintended, such as infinite recursion or out-of-bounds array accesses. For example, Commons-Csv-7 is caused by accessing an out-of-bounds index of an array, which is one of the bug patterns searched for by SpotBugs. Unfortunately, the SpotBugs checker is intra-procedural, while the actual bug computes the array index in one method and then accesses the array in another method. Another example is Lang-49, which causes an infinite loop because multiple methods call each other recursively, and the conditions for stopping the recursion miss a specific input. Both Error Prone and SpotBugs have checkers for infinite loops caused by missing conditions that would stop recursion. However, these checkers target cases that are easier to identify than Lang-49, which would require inter-procedural reasoning about integer values. A third example in this subcategory is Chart-5, which causes an `IndexOutOfBoundsException` when calling `ArrayList.add`. The existing checker for out-of-bounds accesses to arrays might have caught this bug, but it does not consider `ArrayLists`.

On the other hand, the root causes of some bugs are problems that are similar to but not the same as problems targeted by an existing checker. For example, Commons-Codec-8 is about forgetting to override some methods of the JDK class `FilterInputStream`. While SpotBugs and Error Prone have checkers related to streams, including some that warn about missing overrides, none of the existing checkers targets the methods relevant in this bug.

TABLE 2.4: Candidate warnings (W) and bugs (B) obtained from the different automatic matching approaches.

	Approach					
	Diff-based		Fixed warnings		Combined	
Tool	W	B	W	B	W	B
Error Prone	51	33	18	14	53	35
Infer	30	9	14	6	32	11
SpotBugs	51	32	29	22	68	43
Total:	132	74	61	42	**153**	**89**

2.4.7 *Assessment of Methodologies*

We compare the three variants of our methodology and validate that the manual inspection of candidates of detected bugs is crucial.

CANDIDATES OF DETECTED BUGS Our methodology for identifying candidates for detected bugs has three variants (Section 2.2.3.1). Table 2.4 compares them by showing for each variant how many warnings and bugs it identifies as candidates. The number of warnings is larger than the number of bugs because the lines involved in a single bug may match multiple warnings. Overall, identifying candidates based on diffs yields many more warnings, 132 in total, than by considering which warnings are fixed by a bug fix, which yields 61 warnings. Combining the two methodologies by considering the union of candidates gives a total of 153 warnings corresponding to 89 bugs. Since more than one static checker could point to the same bug, the total number of unique candidates for detected bugs by all tools together boils down to 79 bugs.

Figure 2.5 visualizes how the variants of the methodology complement each other. For example, for Error Prone, the fixed warnings-based approach finds 14 candidates, 2 of which are only found by this approach. The diff-based technique finds 21 candidates not found by the fixed warnings approach. Overall, the diff-based and the fixed warnings-based approaches are at least partially complementary, making it worthwhile to study and compare both.

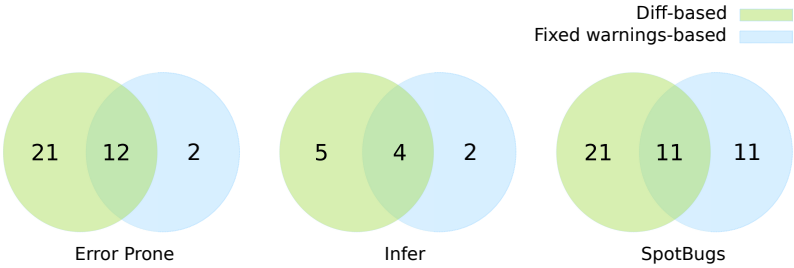


FIGURE 2.5: Candidate detected bugs using the two different automatic matching techniques.

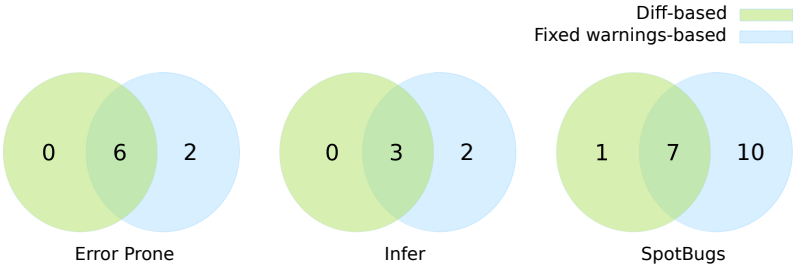


FIGURE 2.6: Actual bugs found using the two different automatic matching techniques.

VALIDATED DETECTED BUGS Figure 2.7 shows how many of the candidates obtained with the diff-based and the fixed warnings-based approach we could validate during the manual inspection. The left chart of Figure 2.7a shows the results of manually inspecting each warning matched by the diff-based approach. For instance, out of the 51 matched warnings reported by Error Prone, 6 are full matches and 2 are partial matches, whereas the remaining 43 do not correspond to any of the studied bugs. The right chart in Figure 2.7a shows how many of the candidate bugs are actually detected by the reported warnings. For example, out of 9 bugs that are possibly detected by Infer, we have validated 3. Figure 2.7b and Figure 2.7c show the same charts for the fixed warnings-based approach and the combined approach.

The comparison shows that the diff-based approach yields many more mismatches than the fixed warnings-based approach. Given this result, one may wonder whether searching for candidates only based on fixed warnings would yield all detected bugs. In Figure 2.6, we see for each

bug detector, how many unique bugs are found by the two automatic matching approaches. For both Error Prone and Infer, although the diff-based approach yields a large number of candidates, the fixed warnings-based methodology is sufficient to identify all detected bugs. For SpotBugs, though, one detected bug would be missed when inspecting only the warnings that have been fixed when fixing the bug. The reason is bug Mockito-11 in Figure 2.2a. The fixed warnings-based methodology misses this bug because the bug fix accidentally re-introduces another warning of the same kind, at line 62 of the fixed code.

In summary, we find that the fixed warnings-based approach requires less manual effort while revealing almost all detected bugs. This result suggests that future work could focus on the fixed warnings-based methodology, allowing such work to manually inspect even more warnings than we did.

MANUAL INSPECTION Table 2.4 shows that the combined approach yields 153 candidate warnings corresponding to 89 (79 unique) bugs. However, the manual validation reveals that only 34 of those warnings and a corresponding number of 31 (27 unique) bugs correspond to actual bugs, whereas the remaining matches are coincidental. Out of the 34 validated candidates, 22 are full matches and 12 are partial matches (Figure 2.7c). In other words, 78% of the candidate warnings and 66% of the candidate bugs are spurious matches, i.e., the warning is about something unrelated to the specific bug and only happens to be on the same line.

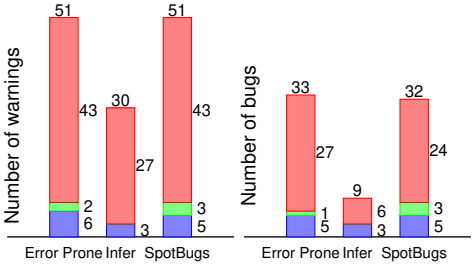
These results confirm that the manual step in our methodology is important to remove coincidental matches. Omitting the manual inspection would skew the results and mislead the reader to believe that more bugs are detected. This skewing of results would be even stronger for bug detectors that report more warnings per line of code, as evidenced in an earlier study [Thu+12].

To ease reproducibility and to enable others to build on our results, full details of all results are available online.[5]

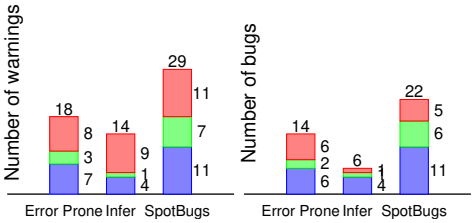
2.5 THREATS TO VALIDITY

As for all empirical studies, there are some threats to the validity of the conclusions drawn from our results. One limitation is the selection of bugs and bug detectors, both of which may or may not be representative for a larger population. To mitigate this threat, we use a large set of real-world

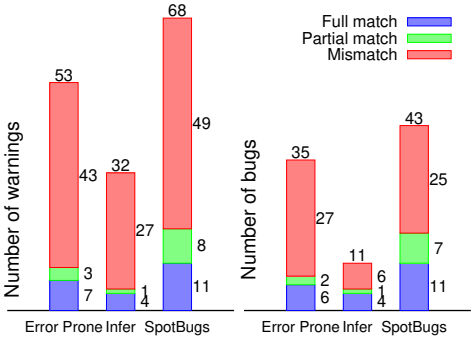
5 https://github.com/sola-da/StaticBugCheckers



(a) Diff-based approach.



(b) Fixed warnings-based approach.



(c) Combined approach.

FIGURE 2.7: Manual inspection of candidate warnings and bugs from the two automatic matching approaches.

bugs from a diverse set of popular open-source projects. Moreover, the bugs have been gathered independently of our work and have been used in previous bug-related studies [Jus+14; Mar+17; Pea+17; Sha+15]. For the bug detectors, we study tools that are widely used in industry and which we believe to be representative for the current state-of-the-art. Despite these efforts, we cannot claim that our results generalize beyond the studied artifacts.

Another threat to validity is that our methodology for identifying detected bugs could, in principle, both miss some detected bugs and misclassify coincidental matches as detected bugs. A reason for potentially missing detected bugs is our assumption that the lines involved in a bug fix correspond to the lines where a developer expects a warning to be placed. In principle, a warning reported at some other line might help a developer to find the bug, e.g., because the warning eventually leads the developer to the buggy code location. Since we could only speculate about such causal effects, we instead use the described methodology. The final decision whether a warning corresponds to a bug is taken by a human and therefore subjective. To address this threat, both authors discussed every candidate for a detected bug where the decision is not obvious.

A final threat to validity results from the fact that static bug detectors may have been used during the development process of the studied projects. If some of the developers of the studied projects use static bug detectors before checking in their code, they may have found some bugs that we miss in this study. As a result, our results should be understood as an assessment of how many of those real-world bugs that are committed to the version control systems can be detected by static bug detectors.

2.6 IMPLICATIONS FOR THIS DISSERTATION AND FUTURE WORK

Our findings regarding the effectiveness of current static bug detectors and the manual inspection of several of the missed bugs pose various implications for this dissertation and more broadly for future research on general bug detection.

The first and perhaps most important is that there is a huge need for bug detection techniques that can detect domain-specific problems. Most of the existing checkers focus on generic bug patterns that occur across projects and often even across domains. However, as most of the missed bugs are domain-specific, more work should complement the existing detectors with techniques beyond checking generic bug patterns. One potential direction

is to consider informal specifications, such as natural language information embedded in code or available in addition to code, e.g., in documentation. The following chapters of this dissertation explore this direction and present several ideas techniques that utilize programs and their documentation to prevent and detect software bugs in novel ways through: (i) Learning from programs source code to infer API concurrency specifications, i.e., documentation, (Chapter 3), (ii) Learning from documentation and runtime behavior to detect inconsistent program behavior (Chapter 4), (iii) Leveraging API documentation to build a subtype checker that detects a novel class of data compatibility bugs (Chapter 5), and (iv) Learning to detect general bug pattern in source code (Chapter 6).

We also suggest that further work on sophisticated yet practical static analysis is required. Given that several currently missed bugs could have been found by inter-procedural variants of existing intra-procedural analyses suggests room for improvement. The challenge here is to balance precision and recall: Because switching to inter-procedural analysis needs to approximate, e.g., call relationships, this step risks to cause additional false positives. Another promising direction suggested by our results is to generalize bug detectors that have been developed for a specific kind of problem to related problems, e.g., `ArrayLists` versus arrays.

Finally, our findings suggest that some bugs are probably easier to detect with techniques other than static checkers. For example, the missed bugs that manifest through clear signs of misbehavior, such as an infinite loop, are good candidates for fuzz-testing with automated test generators.

2.7 CONTRIBUTIONS AND CONCLUSIONS

This chapter investigates how many of all bugs can be found by current state-of-the-art static bug detectors. To address this question, we study a set of 594 real-world Java bugs and three widely used bug detection tools. This is the first study to evaluate the recall of three of the state-of-the-art static bug detectors on a large dataset of real-world bugs.

The main findings of our study are the following:

- The three bug detectors together reveal 27 of the 594 studied bugs (4.5%). This non-negligible number is encouraging and shows that static bug detectors can be beneficial.

- Different bug detectors are mostly complementary to each other. Combining the three studied tools yields an overall bug detection rate of 4.5%.
- The percentage of detected among all bugs ranges between 0.84% and 3%, depending on the bug detector. This result points out a significant potential for improvement, e.g., by considering additional bug patterns. It also shows that checkers are mostly complementary to each other.
- The majority of missed bugs are domain-specific problems not covered by any existing bug pattern. At the same time, several bugs could have been found by minor variants of the existing bug detectors.

3

INFERRING THREAD SAFETY DOCUMENTATION

Thread-safe classes are pervasive in concurrent, object-oriented software. However, many classes lack documentation regarding their safety guarantees under multi-threaded usage, i.e., they lack *concurrency specification*. This lack of documentation forces developers who use a class in a concurrent program to either carefully inspect the implementation of the class, to conservatively synchronize all accesses to it, or to optimistically assume that the class is thread-safe. If a developer makes an uninformed decision about how to use a class in a multi-threaded program, the result would likely be a buggy program or at least, a poor performing one. To overcome the lack of concurrency specification, we present *TSFinder*, an approach to automatically classify classes as supposedly thread-safe or thread-unsafe.

3.1 MOTIVATION

In the previous chapter, we saw that static bug detectors suffer several limitations. One way to tackle some of these limitations is by taking one step back and adopting a preventive approach: *Try to avoid bugs instead of detecting them.* In this chapter, we study and propose a solution to an imminent problem which affects the correctness and performance of any modern software: the lack of API specifications, which at times, could lead to catastrophic failures.

Thread-safe classes are pervasive. They are the corner stone of concurrent, object-oriented programs. A thread-safe class encapsulates all necessary synchronization required to behave correctly when its instances are accessed by multiple client threads concurrently, without additional synchronization from the calling side. Developers of multi-threaded object-oriented programs often rely on thread-safe classes to cast away the burden of ensuring the thread safety of their applications.

Unfortunately, it is not always clear to a developer who uses a class whether the class is thread-safe or not. The reason is that many classes do not provide any or only partial information about their thread safety. Instead, it is common to find questions on web forums, such as Stack Overflow, about the thread safety of a specific class. For example, one developer asked about the thread safety of the widely used `javax.xml.parsers.DocumentBuilder` class.[1] Another developer questioned the thread safety of the crucial JDK class `java.util.Random`.[2] Developers often complain about the lack of thread safety documentation. For instance, the developer who reported that earlier versions of JDK format classes are not thread-safe notes that: "Not being thread-safe is a significant limitation on a class, with potentially dire results, and not documenting the classes as such is dangerous."[3] Eventually, the accepted fix was to explicitly state in the documentation that JDK format classes are not thread-safe. Another developer complains about the lack of thread safety documentation of the classes `java.beans.PropertyChangeSupport` and `java.beans.VetoableChangeSupport` and writes in her bug report: "[...] However, the documentation does not indicate either their thread-safety (sic) or lack thereof. In keeping with the current documentation standards, this point should be indicated in the class documentation. This will allow implementors to benefit from any thread-safety (sic) of the synchronization in the class implementations and allow proper multi-threaded implementation of these two classes"[4]

The lack of adequate documentation about the thread safety of classes has several negative consequences. First, a developer may solve the problem by manually analyzing the classes she wants to reuse. However, this approach spoils some of the benefits of reusing an existing class because it forces the developer to inspect and understand the class implementation, breaking the encapsulation provided by the class API. Second, a developer may conservatively assume that a class is not thread-safe and carefully synchronize all concurrent accesses to the class to avoid concurrency bugs, such as data races, atomicity violations, and deadlocks. However, if the class is already thread-safe, this additional synchronization imposes additional runtime overhead and may unnecessarily limit the level of parallelism achieved by the program. Finally, a developer may optimistically assume a class to be thread-safe. However, if the class turns out to not provide this guarantee, the

1 https://www.stackoverflow.com/questions/56737
2 https://www.stackoverflow.com/questions/5819638
3 https://bugs.java.com/view_bug.do?bug_id=4264153
4 https://bugs.java.com/view_bug.do?bug_id=5026703

program may suffer from concurrency bugs, e.g., [Tsba; Tsbb; Tsbc], which often become apparent only under specific interleavings and therefore may easily remain unnoticed during testing. In all three scenarios, the developer takes a poorly guided decision that relies on her limited understanding of an implementation or on luck.

This chapter addresses the problem of missing thread safety documentation by automatically classifying a given class as thread-safe or thread-unsafe. Our approach, called TSFinder, is a statistical, graph-based learning technique that learns from a relatively small set of classes known to be thread-safe or thread-unsafe the distinguishing properties of these two kinds of classes. The approach is enabled by two contributions. First, TSFinder uses a lightweight static analysis of the source code of the class to extract information and represent this information in a graph. Second, we use graph-based classification techniques – graph kernels [Vis+10] combined with support vector machines (SVM) [SS02] to learn a classifier for previously unseen classes. TSFinder helps developers assess the thread safety of an otherwise undocumented class, enabling a developer to take an informed decision on whether and how to use the class.

Our work is complementary to techniques for finding concurrency bugs, which has been extensively studied in the past [AHB03; Cho+02; FF04; FF09; Lu+06; LC09; OC03; PG01; Sav+97; WS06; XBH05], in particular in the context of thread-safe classes [CLP17; Nis+12; PG12; SR14; SR15; SRJ15; TC16]. These approaches consider supposedly thread-safe classes and try to find corner cases in their implementation that a developer has missed. Instead, TSFinder addresses classes for which it is unknown whether the class is even supposed to be thread-safe and tries to answer that question in an automatic way. Applying existing bug detection techniques to answer this question would likely result in missing thread-unsafe classes (by testing-based approaches) or missing thread-safe classes (by sound static analyses). Our work also relates to existing work on inferring [ABL02; HRD07] and improving [Jia+17; McB+17; TR16; Zho+17] documentation. We extend this stream of work to concurrency-related documentation, which has not yet been studied.

In Section 3.2, we give an overview of TSFinder. Sections 3.3 and 3.4 fill in the details. Sections 3.5 and 3.6 summarize the implementation and evaluation. Section 3.7 discusses the limitations of TSFinder. Finally, in Section 3.8 we conclude the chapter and discuss future work.

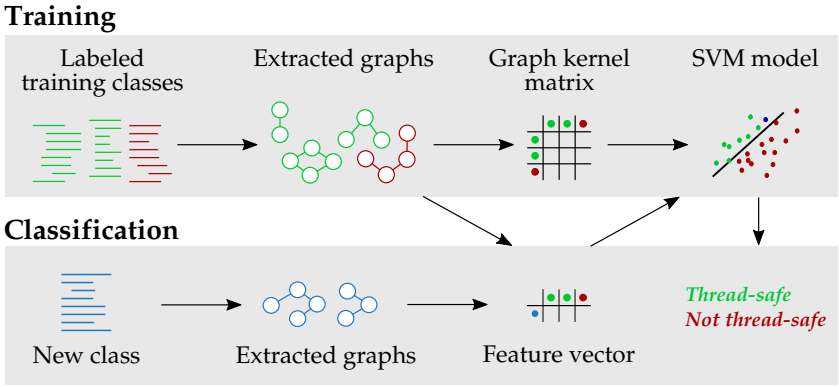


FIGURE 3.1: Overview of TSFinder: Inferring thread safety using static analysis and graph kernels.

3.2 CHALLENGES AND OVERVIEW

The goal of this work is to automatically document classes as supposedly thread-safe or thread-unsafe. The approach should be efficient enough to scale to hundreds of classes, e.g., all classes in a 3rd-party library, and accurate enough to provide reliable documentation. Achieving this goal is challenging for several reasons. First, there are different approaches for ensuring that a class is thread-safe, e.g., making the class immutable, using language-level synchronization primitives, building on other thread-safe classes, using lock-free data structures, and combinations of these approaches. Because of this diversity, a simple check, e.g., for whether a class has synchronized methods, is insufficient to determine thread safety. Second, the thread safety of a class may depend on other classes. In particular, inheriting from a thread-unsafe class may compromise the thread safety of the child class. Third, extensive reasoning about concurrent behavior, e.g., to reason about different interleavings [Vis+03], can easily require large amounts of computational resources, which conflicts with our scalability goal.

Figure 3.1 provides an overview of our approach to infer thread safety documentation. The approach consists of a training phase, where it learns from a set of classes known to be thread-safe and thread-unsafe, and a prediction phase, where it infers thread safety documentation for a previously unseen class. Both phases combine a lightweight static analysis that extracts graph representations of classes with a graph-based classifier.

The graph-based classification converts graphs into vectors by computing the similarity between graphs of a to-be-classified class and graphs in the trained model. These vectors are then classified using a model based on support vector machines (SVM). The following illustrates the main steps of TSFinder using the Java class in Figure 3.2a.

EXTRACTING FIELD-FOCUSED GRAPHS To apply machine learning to the thread safety classification problem, we need to represent classes in a suitable form. Our approach exploits the structured nature of programs by representing a class as a set of graphs. Since multi-threading is mainly about sharing and allowing multiple concurrent accesses to resources, the graphs represent shared resources and how these resources are accessed.

For the example class, Figure 3.2b shows the graphs extracted by TSFinder. Each graph focuses on a single field or a combination of fields of the class. The graphs represent read and write accesses to the fields, call relationships between methods, and the use of synchronization primitives, such as the `synchronized` keyword. For example, the first graph in Figure 3.2b which focuses on the `seq` field shows that the field is read by the `isMax` method, written by the `reset` method, and both read and written by the `next` method. Furthermore, the graph represents the call relationship between `next` and `isMax`.

COMPUTING GRAPH KERNELS After extracting a set of graphs for each class under analysis, TSFinder checks for each graph whether it is similar to graphs that come from thread-safe or from thread-unsafe classes. To this end, we use the graph kernels [Vis+10], i.e., mathematical functions that compute the pairwise similarity of graphs. TSFinder computes the similarity of each graph of a class and the graphs of classes known to be thread-safe or thread-unsafe. The similarity values yield a vector of numbers, called the *graph vector* or *embedding*. For the running example, the approach computes three graph vectors, one for each graph, as illustrated in Figure 3.2c.

LEARNING A CLASSIFICATION MODEL To train a classifier that can distinguish thread-safe classes from thread-unsafe classes, TSFinder trains a model using a corpus of classes with known thread safety. The approach combines all graph vectors of a class into a single vector, called *class vector*, that represents the entire class (Figure 3.2d) along with a label denoting whether the class is thread-safe or not. Finally, the labeled class vectors

are used to train a classification model that distinguishes between the two kinds of classes.

CLASSIFYING A NEW CLASS Given a new class, our approach extracts graphs and computes a class vector as in the previous step. Based on the trained model, TSFinder then classifies the class by querying the model with this vector. For the example in Figure 3.2, TSFinder infers that the class is *thread-safe* and adds this information to the class documentation.

3.3 EXTRACTING FIELD-FOCUSED GRAPHS

The first step of our approach is to extract graphs from classes via a lightweight static analysis. This section explains the properties extracted by the static analysis (Section 3.3.1) and how we summarize these properties into graphs (Section 3.3.2).

3.3.1 *Static Analysis*

TSFinder performs a lightweight static analysis that extracts various properties of a class under analysis. We focus on two kinds of properties: *unary properties*, which describe program elements of the class, and *binary properties*, which describe relationships between program elements and properties of program elements. We choose properties relevant for concurrency, e.g., memory locations, accesses to memory locations, and memory visibility guarantees of these accesses.

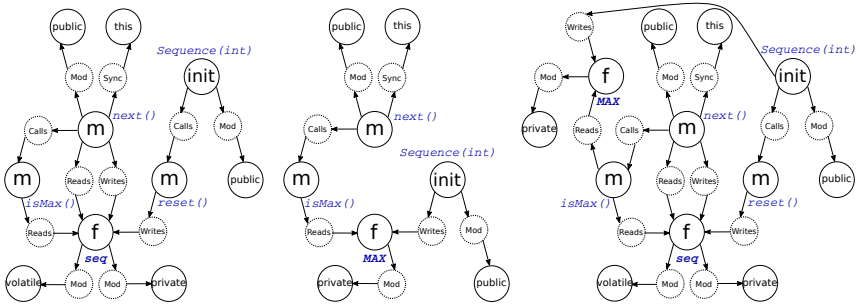
UNARY PROPERTIES The static analysis extracts the following unary properties from each class:

Definition 3.3.1 (Unary properties) *Let C be the class under analysis. Let C_f be the set of fields, C_m be the set of methods, and C_{const} be the set of class constructors and static constructors defined by C. The set of unary properties of C is:*

$$C_{unary} = C_f \cup C_m \cup C_{const}$$

```java
public class Sequence {
  private volatile int seq;
  private int MAX;

  public Sequence(int m) {
    MAX = m;
    reset();
  }

  public synchronized int
        next() {
    if(!isMax())
      return seq++;
    return -1;
  }

  boolean isMax() {
    return seq > MAX;
  }

  void reset() {
    seq = 0;
  }
}
```

(a) Java class.



(b) Extracted graphs. The graphs from left to right correspond to fields `seq`, `MAX`, and the pair (`seq`, `MAX`), respectively. The identifier names, in italic and blue font, are not used for classification, but shown only for illustration.

g_1	g_2	$\cdots$	$g_{4,860}$	g_1	g_2	$\cdots$	$g_{4,860}$	g_1	g_2	$\cdots$	$g_{4,860}$
[0.350	0.436	…	0.573]	[0.355	0.536	…	0.584]	[0.392	0.588	…	0.567]

(c) Vectors of the three graphs in Figure 3.2b. The trained model has 4,860 graphs.

$min(g_1)$	$max(g_1)$	$avg(g_1)$	$min(g_2)$	$max(g_2)$	$avg(g_2)$	…	$avg(g_{4,860})$
[0.350	0.392	0.366	0.436	0.588	0.520	…	0.575]

(d) Class vector of the entire class.

FIGURE 3.2: A Java class and graphs extracted from it by our analysis. TSFinder predicts this class to be *thread-safe*.

For example, our approach extracts the following set of unary properties from the class in Figure 3.2a:

$$C_f = \{seq, MAX\}, C_m = \{next(), isMax(), reset()\}$$

and

$$C_{const} = \{Sequence(int)\}$$

BINARY PROPERTIES To capture relationships between different program elements and properties of program elements, the analysis extracts several binary properties:

Definition 3.3.2 (Binary properties) *Let C be the class under inspection, and C_{const}, C_m, C_f as defined above. We define the following binary relations Rels:*

- $Calls : \{C_{const} \cup C_m\} \times \{C_{const} \cup C_m\}$
- $Reads : \{C_{const} \cup C_m\} \times \{C_f\}$
- $Writes : \{C_{const} \cup C_m\} \times \{C_f\}$
- $Sync : \{C_m\} \times \{this, lock\}$
- $Mod : \{C_{const} \cup C_m \cup C_f\} \times \{public, protected, private, static, volatile, final\}$

The set of binary properties of C is:

$$C_{binary} = Calls \cup Reads \cup Writes \cup Sync \cup Mod$$

The binary properties capture a rich set of relations relevant to our thread safety prediction task, e.g., whether a method is public, what fields a method reads and writes, and whether a method is synchronized. The set $\{this, lock\}$ represents objects that the class uses as locks, where *this* represents a self-reference to the current instance and *lock* represents any other object.

For our running example in Figure 3.2a, the binary properties include that the public class constructor `Sequence(int)` writes to the field `MAX`, that the method `next()` reads and writes the field `seq`, and that the method `next()`is `synchronized` on `this`. Note that the absence of properties also conveys information. For example, the absence of a binary relation $(MAX, volatile) \in Mod$ indicates that the `MAX` field is non-volatile.

FLATTENING THE CLASS HIERARCHY The thread safety of a class not only depends on its own implementation, but also on the implementation of its superclasses. E.g., a class may inherit a method that does not synchronize data accesses and therefore become thread-unsafe, even though the subclass alone would be thread-safe [PG13]. Our static analysis addresses this challenge by flattening the class hierarchy. Specifically, the analysis recursively merges the unary and binary properties of each class with those of its superclass until reaching the root of the class hierarchy. The merging follows the inheritance rules of the Java language. For example, the properties related to a superclass method that is not overridden by the subclass are merged into the properties of the subclass.

3.3.2 *Field-focused Graphs*

Given the properties extracted by the static analysis, TSFinder summarizes this information into a set of graphs for each class. Traditionally, programs have been represented by a variety of graphs suited for different purposes. For example, abstract syntax trees, control-flow graphs, and program dependency graphs have been used to analyze the syntax, control flow, and data flow of programs. The following presents two kinds of graphs designed specifically to reason about concurrency-related properties of classes. The basic idea is to summarize in each graph how clients of the class may access a field or a combination of fields of the class. We call these graph representations *field-focused graphs*.

Before presenting field-focused graphs, we define a single graph per class, which conflates all properties known about this class:

Definition 3.3.3 (Class graph) *Given a class C, let C_{unary} and C_{binary} be the unary and binary properties of C, respectively. The class graph of C is a directed multi-graph $g_C = (V_C, E_C)$, where $V_C = V_{Rels} \cup C_{unary} \cup \{this, lock, public, protected, private, static, volatile, final\}$ are vertices that represent program elements and properties of them, and $V_{Rels} = \{Calls, Reads, Writes, Sync, Mod\}$ are special nodes that represent the different relations in C_{binary}. Each special node is labeled with the name of the relation, i.e., with Calls, Reads, Writes, Sync, or Mod and is connected to its binary operands by the set E_C of directed unlabeled edges.*

One possible approach would be to predict the thread safety of a class based on its class graph. However, most class graphs are dissimilar from most other class graphs, independently of whether the classes are thread-

safe, because classes and therefore also their class graphs are very diverse. An important insight of our work is that this problem can be addressed by deriving smaller graphs from the class graph, so that each small graph captures a coherent subset of concurrency-related properties. The intuition is that these smaller graphs capture recurring implementation patterns of thread-safe and thread-unsafe classes, enabling TSFinder to learn to distinguish them.

TSFinder derives smaller graphs from the class graph by focusing on a single field or a combination of fields:

Definition 3.3.4 (Field-focused graph) *Given a non-empty subset $F \subseteq C_f$ of the fields of a class C and a class graph g_C where $g_C = (V_C, E_C)$, the field-focused graph $g_F = (V_F, E_F)$ contains all vertices reachable from F, i.e., $V_F = \{v \mid \exists\, v_f \in F \text{ s.tt. } reachable_{g_C}(v_f, v) \text{ and } reachable_{g_C}(v, v_f)\}$, and contains all edges connecting these vertices.*

For a directed graph $g = (V, E)$ where u and $v \in V$,

$$reachable_g(u, v) \iff \text{there exists a directed edge from } u \text{ to } v.$$

If the set F contains a single field, then the field-focused graph captures all program elements related to this field, as well as the relations between them. Such a single-field graph summarizes how clients of the class may access the field and to what extent these accesses are protected by synchronization.

For the example in Figure 3.2a, TSFinder extracts two graphs that focus on single fields, shown as the first two graphs in Figure 3.2b. They focus on the fields `seq` and `MAX`, respectively.

Some characteristics of thread-safe classes cannot be captured by single-field graphs. For example, a thread-safe class may update two semantically related fields together and use a single lock or a synchronized method to protect the access to these fields. TSFinder captures such behavior by also considering sets F of multiple fields, which yields multi-field graphs. Specifically, the approach considers all pairs of fields for which there exists at least one method that reads or writes from both fields. To bound the overall number of graphs extracted per class, we focus on field-focused graphs with $|F| \leq 2$, i.e., single fields or pairs of fields.

As an example of a multi-field graph, consider the third graph in Figure 3.2b. Because the class method `isMax()` reads both fields, the approach extracts a graph that captures both fields together.

Intuitively, the reason why field-focused graphs are effective at characterizing the thread safety of a class is that they capture various patterns for

making a class thread-safe. Whether a class is thread-safe depends on how the class accesses its internal state, i.e., its fields, and in what ways these accesses are protected by synchronization. Field-focused graphs capture the various ways to implement thread safety, e.g., using synchronized methods, volatile fields, or by making a class immutable. By capturing these implementation patterns, the graphs enable TSFinder to determine whether a class is thread-safe.

GRAPH CANONICALIZATION The final step of extracting field-focused graphs from classes is to canonicalize the graphs. The motivation is that, to learn recurring patterns in implementations of thread-safe and thread-unsafe classes, the extracted graphs need to be comparable across different classes. In particular, they should not contain identifier names, such as method and field names, as these vary across different classes and projects. Therefore, our approach renames each node that represents a method to `m`, while two special node names `init` and `clinit` are reserved for class constructors and static constructors, respectively. Similarly, all fields are renamed to `f`.

3.4 CLASSIFYING CLASSES

Classifying graphs is a classical problem in several domains such as bio- and chemo-informatics [Bor+05; Ral+05; Swa+05], image analysis [HB07], and web and social network analysis [Vis+14]. Traditional approaches to this problem [Vis+10] use a so-called kernel method [SS02], a function to compute the similarity between two graphs. The pairwise similarities between graphs are then used as vector embeddings to represent the graphs for classification.

We adopt a variant of this approach to our problem of classifying thread-safe classes. TSFinder first builds several graphs per class (Section 3.3.2). It then uses the kernel method through a graph kernel function to generate vectors (embeddings) for graphs (Section 3.4.2.1). Instead of training a machine learning model on several individual graphs from each class, we combine embeddings of graphs extracted from the same class into one single embedding per class for the machine learning model to learn (Section 3.4.2.2). This step allows TSFinder to classify thread-safe classes using all generated graphs from a class.

Based on the field-focused graphs extracted for each class, TSFinder learns how to classify classes into supposedly thread-safe and thread-unsafe

classes. To this end, the approach combines a graph kernel, which computes the similarity of two graphs, with a SVM, which classifies each class based on the similarity of its graphs to other graphs from classes known to be thread-safe or not.

The basic idea is to perform three steps:

1. Given a class, compare its graphs to graphs of classes known to be thread-safe or thread-unsafe. For each pair of graphs, compute a similarity score and summarize all scores into a vector per graph.

2. Combine all graph vectors of a class into one single class vector that summarizes the similarity of graphs extracted from the class to graphs in the trained model.

3. Classify a class by querying a vector-based binary classifier using the resulting class vector. The classifier has been trained with the class vectors of the classes with known thread safety.

The remainder of this section presents these steps in detail.

3.4.1 *Background: Graph Kernels*

Checking whether two graphs are isomorphic is a computationally hard problem for which no polynomial-time algorithm is known. In contrast, graph kernels offer an efficient alternative that compares graph substructures in polynomial time. In essence, a graph kernel is a function that takes two graphs and yields a real-valued similarity score. Given a list of graphs $g_1, .., g_n$ and a kernel k, one can compute a matrix $K = (k(g_i, g_j))_{i,j}, 1 \leq i, j \leq n$, that contains all pairwise similarity scores of the graphs. This matrix, called the *kernel matrix*, is symmetric and positive-definite.

In this work, we build upon a fast, scalable, state of the art kernel, the *Weisfeiler-Lehman (WL) graph kernel* [She+11]. It is based on the Weisfeiler-Lehman graph isomorphism test [WL68], which augments each labeled node by the sorted set of its direct neighbors and compresses this augmented label into a new label. This step is repeated until the sets of node labels of the two graphs are different or until reaching a maximum number of iterations h. Given a graph g, we refer to the sequence of graphs obtained by this augmentation and compression step as $g_0, g_1, .., g_h$, where $g_0 = g$ and g_h is the maximally augmented and compressed graph. We call this sequence of graphs the *WL sequence* of g.

Given two graphs and their WL sequences, we compute the graph kernel as follows:

Definition 3.4.1 (Weisfeiler-Lehman kernel) *The graph kernel of g and g' is*

$$k(g, g') = k_{sub}(g_0, g'_0) + k_{sub}(g_1, g'_1) + \ldots + k_{sub}(g_h, g'_h)$$

The function k_{sub} is a subtree kernel function.

Definition 3.4.2 (Weisfeiler-Lehman subtree kernel) *The subtree graph kernel of g and g' is*

$$k_{sub}(g, g') = \langle \phi(g), \phi(g') \rangle$$

where the notation $\langle ., . \rangle$ denotes the inner product of two vectors.

The ϕ function vectorizes a labeled graph by counting the original and compressed node labels of the graphs in the WL sequences of g and g'. Specifically, let Σ_i be the set of node labels that occur at least once in g or g' at the end of the i-th iteration of the algorithm, and let $c_i(g, \sigma_{ij})$ be the number of occurrences of the label $\sigma_{ij} \in \Sigma_i$ in the graph g. Based on the counter c_i, we compute ϕ as follows:

$$\begin{aligned}\phi(g) = (&c_0(g, \sigma_{01}), \ldots, c_0(g, \sigma_{0|\Sigma_0|}), \ldots, \\ &c_h(g, \sigma_{h1}), \ldots, c_h(g, \sigma_{h|\Sigma_h|}))\end{aligned}$$

3.4.2 *Training*

The goal of the training step of TSFinder is to create a binary classification model that predicts whether a given class is thread-safe or thread-unsafe. We use a supervised learning technique and therefore require training data. As training data, we use two sets of classes C_{TS} and $C_{\widetilde{TS}}$, which consist of known thread-safe and known thread-unsafe classes, respectively. For each of these classes, the static analysis (Section 3.3) extracts a set of graphs.

3.4.2.1 *Graphs Vectors*

TSFinder first computes a vector representation of each graph based on the graph kernel function in Definition 3.4.1. Intuitively, the vector characterizes a graph by summarizing how similar it is to other, known graphs in the training data.

More technically, the approach computes the vector representation of a graph in three steps:

1. Fix the order of all graphs in $G_{C_{TS} \cup C_{\widetilde{TS}}}$ to obtain a list of graphs $g_1, .., g_n$. The specific order does not matter, as long as it remains fixed.

2. Compute the kernel matrix of all graphs $K = (k(g_i, g_j))_{i,j}$ for $1 \leq i, j \leq n$.

3. For each graph g_i, the i-th row of K is the vector representation of g, called graph vector.

3.4.2.2 *Combining Class Graphs*

Given the graphs vectors of a class, we combine these vectors into a single class vector. Intuitively, the class vector should summarize to what extent the individual graphs of a class resemble the graphs of classes in the training data. If a class has graphs that are very similar to graphs that typically occur in thread-safe classes, then the class is more likely to thread-safe. Likewise, a class with graphs that mostly resemble graphs from thread-unsafe classes is more likely to also be thread-unsafe. To encode this intuition, we create a class vector by computing the minimum, maximum, and average similarity of all the graphs of the class against all graphs extracted from the training classes.

Let $n = |G_{C_{TS} \cup C_{\widetilde{TS}}}|$ be the total number of graphs extracted from all classes in the training data. For a specific class C, let G_C be the set of all graphs TSFinder extracted from C and $m = |G_C|$ be the total number of these graphs. For each graph $g_i \in G_C$ where $1 < i < m$, let $f^j_{g_i}$ where $1 < j < n$ be the jth feature of graph g_i of the class C. Our approach computes the class vector by calculating $\forall j \in 1, \ldots, n$:

$$min(f^j_{g_i} \ \forall i \in 1, \ldots, m),$$
$$max(f^j_{g_i} \ \forall i \in 1, \ldots, m),$$
$$mean(f^j_{g_i} \ \forall i \in 1, \ldots, m)$$

and concatenating these $n * 3$ values into a single vector.

For example, the class vector in Figure 3.2d has 3*4860=14580 elements. The first three elements are the minimum, maximum, and mean similarity of the graphs in Figure 3.2b compared to the first graph in the list of graphs extracted from the training classes. The next three elements are the minimum, maximum, and mean similarity of the graphs in Figure 3.2b compared to the second graph extracted from the training classes, ... etc.

3.4.2.3 *Classifier*

Given the class vectors and their corresponding labels $l_1, .., l_n$ that indicate whether a class c is from C_{TS} or from $C_{\widetilde{TS}}$, we finally feed the labeled vectors into a traditional vector-based classification algorithm. By default, TSFinder uses a SVM for learning the classifier. Our evaluation also considers alternative algorithms.

3.4.3 *Classifying a New Class*

Once TSFinder has learned a model, we use it to predict the thread safety of a new class. Let C_{new} be the new class for which we wish to infer its supposed behavior regarding thread safety. The approach computes a class vector of C_{new} in the same way as for training. At first, TSFinder extracts field-focused graphs from C_{new}, which yields a set $G_{C_{new}}$ of graphs. For each graph $g \in G_{C_{new}}$ the approach computes the graph vector of g by computing its graph kernel against all graphs in our training data:

$$vec(g) = (k(g, g_j))_{1j}, j = 1, 2, \ldots, n$$

where $g_j \in G_{C_{TS} \cup C_{\widetilde{TS}}}$ is the set of graphs in the learned model and $n = |G_{C_{TS} \cup C_{\widetilde{TS}}}|$ is the total number of graphs in the model. Given the set of graphs vectors, TSFinder combines these graphs into a single class vector as described in Section 3.4.2.2 and queries the trained model to obtain a classification label for the class C_{new}. The label indicates whether the model predicts the class to be thread-safe or thread-unsafe.

3.5 IMPLEMENTATION

We implement TSFinder into a fully automated tool to analyze Java classes. The static analysis builds on the static analysis framework Soot [VR+99]. Given a class, either as source code or byte code, the analysis extracts field-focused graphs by traversing all program elements, by querying the call graph, and by analyzing definition-use relationships of statements. We use the GraphML format [Bra+02] to store graphs. To compute the WL graph kernel, we build on an existing Python implementation [She+11]. The SVM model is implemented on top of the Weka framework [Fra+05]. Our implementation and dataset of thread-safe and thread-unsafe classes are available as open-source.[5]

5 https://github.com/sola-da/TSFinder

3.6 EVALUATION

The evaluation is driven by four main research questions:

- RQ_1: How many classes come with documentation about their thread safety? (Section 3.6.1)
- RQ_2: How effective is TSFinder in classifying classes as thread-safe or thread-unsafe? (Section 3.6.2)
- RQ_3: How efficient is TSFinder? (Section 3.6.3)
- RQ_4: How does TSFinder compare to variants of the approach and to a simpler approach? (Section 3.6.4)

3.6.1 *RQ_1: Existing Thread Safety Documentation*

To better understand the current state-of-the art in documenting thread safety, we systematically search all 179,239 classes from the Qualitas corpus for thread safety documentation. We focus on documentation provided as part of the Javadoc comments of a class and its members, and ignore any other documentation, e.g., on project web sites or in books. Most real-world classes have Javadoc documentation and it is a common software engineering practice to document class-level properties, such as thread safety, there.

Our inspection proceeds in two steps. At first, we generate the Javadoc HTML files for all classes and automatically search them for keywords related to concurrency and thread safety. Specifically, we search for "concu", "thread", "sync", and "parallel". We choose these terms to overapproximate any relevant documentation. In total, the search yields hits in 8,655 of the 179,239 classes.

As the second step, we manually analyze a random sample of 120 of the 8,655 classes. For each sampled class, we inspect the Javadoc and search for any documentation related to the thread safety of the class. Based on this inspection, we classify the class in one of the following four categories.

DOCUMENTED AS THREAD-SAFE The documentation explicitly specifies that the class is supposed to be thread-safe or this intention can be clearly derived from the available information. Examples include:

- The class-level documentation states "This class is thread-safe".

- The name of the class is `SynchronousXYChart` and the project also contains a class `XYChart`, indicating that the former is a thread-safe variant of the latter.
- The class-level documentation states "Mutex that allows only one thread to proceed [while] other threads must wail until the one finishes". The semantics of a mutex implementation imply that the class is thread-safe because mutexes are accessed concurrently without acquiring any additional locks.

DOCUMENTED AS THREAD-UNSAFE The documentation explicitly specifies that the class is not supposed to be thread-safe or this intention can be clearly derived from the available information. Examples include:

- The class-level documentation states "This class is not thread-safe" or "not to be used without synchronization".
- The class-level documentation states "We are not using any synchronized so that this does not become a bottleneck".
- The class-level documentation states "The class (..) shall be used according to the Swing threading model", which implies that only the Swing thread may access instances of the class and that the class is not thread-safe [WO04].

DOCUMENTED AS CONDITIONALLY THREAD-SAFE The documentation specifies the class to be thread-safe under some condition. Examples include:

- The class depends on another class and has the same thread safety as this other class.
- All static methods of the class are thread-safe, whereas non-static methods are not necessarily thread-safe.

NO DOCUMENTATION ON THREAD SAFETY The documentation does not mention thread safety and we cannot derive from other available information whether the class is supposed to be thread-safe. Examples of documentation that matches our search terms but does not document thread safety include:

- The class implements a graph data structure and its documentation says that it "permits *parallel* edges".

TABLE 3.1: Existing thread safety documentation.

Documented as:	Number	Percentage
Thread-safe	11	9.2%
Not thread-safe	12	10.0%
Conditionally thread-safe	2	1.7%
No documentation	**95**	**79.2%**
Total inspected	120	100.0%

- The method-level documentation specifies that an argument or the return value of the method is supposed to be thread-safe. While such a statement is about thread safety, it does not specify this property for the current class.

Table 3.1 summarizes the results of this classification. We find that most (79.2%) of the inspected classes do not document the thread safety of the class, but hit our search terms for some other reason. In the documented subset of classes, which sums up to 20.8%, roughly the same number of classes is documented as thread-safe and thread-unsafe, respectively.

Under the assumptions that our search terms cover all possible thread safety documentation and that the 120 sampled classes are representative for the entire population of classes in the corpus, we can estimate the percentage of documented classes in the corpus:

$$\frac{\%\ documented\ *\ Search\ hits}{Total\ classes} = \frac{0.208 * 8,655}{179,239} = 1.004\%$$

In summary, the vast majority of real-world Java classes do not document whether they are thread-safe or not. Among the few documented classes, 47.8% and 52.2% are documented as thread-safe and thread-unsafe, respectively. We conclude that the current state of thread safety documentation is poor and will benefit from automatic inference of documentation.

3.6.2 *RQ_2: Effectiveness of TSFinder*

3.6.2.1 *Dataset and Graph Extraction*

For the remaining evaluation, we use a set of 230 classes gathered from JDK version 1.8.0_152. These classes are documented to be either thread-safe or thread-unsafe, providing a ground truth for our evaluation. Table 3.2 shows the number fo fields, methods, and lines of code of these classes. In total, the classes sum up to 74,313 lines of Java code. The last three columns of Table 3.2 provide statistics about the graphs that TSFinder extracts. On average, the static analysis extracts 21.1 graphs per class, which yields a total of 4,860 graphs that the approach learns from.

Although the number of thread-safe and thread-unsafe classes is equal, the total number of extracted graphs from thread-unsafe classes is about 1.4 the number of graphs extracted from thread-safe classes. Since TSFinder uses the entire set of 4,860 graphs to construct the class vector for any class, this imbalance does not prevent the approach from learning an effective classifier. The number of graphs per category in Table 3.2 is disproportionate to the number of fields and methods in the same category due to flattening the class hierarchy (Definition 3.3.1).

TABLE 3.2: Classes and extracted field-focused graphs used for training and cross-validation.

		Fields			Methods			LoC			Extracted graphs		
Classes	Count	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Graphs	Vertices	Edges
Thread-safe	115	1	64	8.7	2	163	34.7	13	4,264	430.2	1,989	128,493	150,850
Thread-unsafe	115	0	55	4.3	1	103	23.8	7	1,931	219.7	2,871	151,410	170,473
All	230	0	64	6.4	1	163	29.2	7	4,264	323.1	4,860	279,903	321,323

TABLE 3.3: Effectiveness of classification via 10-fold cross validation across 230 classes with $h = 3$.

	Thread-safe		Thread-unsafe	
Accuracy	Precision	Recall	Precision	Recall
94.5%	94.9%	94.0%	94.2%	95.0%

3.6.2.2 *Results*

To evaluate the effectiveness of TSFinder, we apply it to the 230 classes and measure precision, recall, and accuracy. We perform 10-fold cross validation, a standard technique to evaluate supervised machine learning. The technique shuffles and splits all labeled data, i.e., our 230 thread-safe and thread-unsafe classes into ten equally sized sets. For each set, it then trains a model with the classes in the other nine sets and predicts the labels of the remaining classes using the trained model. We measure accuracy as the percentage of correct classifications among all classifications made by TSFinder. We measure precision and recall both for predicting thread safety and for predicting thread unsafety. With respect to thread (un)safety, precision means the percentage of correct thread (un)safety predictions among all predictions saying that a class is thread-(un)safe. Recall means the percentage of classes classified as thread-(un)safe among all classes that are actually thread-(un)safe.

Table 3.3 shows the results of the 10-fold cross validation. The classification accuracy is 94.5%, i.e., TSFinder correctly predicts the thread safety of the vast majority of classes. The precision and recall results allow the reader to further understand how incorrect predictions are distributed. For example, the fact that the precision for thread safety is 94.9% means the following: When the approach predicts a class to be thread-safe, then this prediction is correct in 94.9% of the cases. Similar, the recall for thread-safety of 94.0% means that TSFinder finds 94.0% of all thread-safe classes and misses the remaining 6%.

3.6.2.3 *Manual Inspection*

To better understand the limitations of TSFinder, we inspect some of the mis-classified classes.

THREAD-SAFE CLASS PREDICTED AS NOT THREAD-SAFE TSFinder mistakenly predicts the thread-safe `ConcurrentLinkedQueue` class to be thread-unsafe. This queue implementation builds upon a non-blocking algorithm [MS96]. Since our training set includes only six classes that use a similar lock-free implementation, the training data may not be sufficient for the classifier to generalize to the `ConcurrentLinkedQueue` implementation. Nevertheless, TSFinder correctly predicts some of the other classes that use non-blocking implementations.

THREAD-UNSAFE CLASS PREDICTED AS THREAD-SAFE The approach predicts `TreeSet` and `EnumSet` as thread-safe, even though they are thread-unsafe implementations of the abstract class `AbstractSet`. We suspect these misclassification to be due limitations of the the learned model to generalize to previously unseen cases.

INACCURATE DOCUMENTATION TSFinder classifies the class `PKIXCertPathValidatorResult` as thread-safe, even though its documentation labels it as not thread-safe. Manually inspecting the implementation shows that the class is indeed thread-safe. The private fields of the class are initialized by the constructor and after that cannot be written to. This case illustrates that TSFinder can not only add otherwise missing documentation, but could also be useful for validating existing documentation.

> In summary, our classifier correctly predicts the thread safety of a class in 94.5% of the cases. The precision and recall for identifying thread-safe classes are 94.9% and 94.0%, respectively. We conclude that the approach achieves its goal of automatically and precisely identifying whether a class is supposed to be thread-safe.

3.6.3 *RQ_3: Efficiency of TSFinder*

We evaluate the efficiency of our approach by measuring the time required for the different steps. All experiments are performed on a machine with 4 Intel i7-4600U CPU cores and 12GB of memory. Training the classifier with a set of training classes is a one-time effort. For the 230 training classes that we use in this evaluation, the training takes approximately 11.7 minutes, including all computation steps, such as extracting graphs, computing graph kernels, and training the SVM model. When querying TSFinder with

TABLE 3.4: Effect of the WL kernel iterations parameter h on classification.

h	1	2	3	4	5	6	7
Accuracy	89.7%	94.1%	94.5%	94.4%	93.9%	94.1%	94.1%

a new class, the approach extracts graphs from this class and classifies the class based on the graphs. The former step takes about 3 seconds and it dominates the latter which is negligible, on average over all 230 training classes.

TSFinder stores graphs extracted from training classes as part of its trained model. These graphs are used to compute the pairwise similarity of graphs extracted from the class under inspection to build the vector embedding of the class. For our model trained with 230 graphs, the total size of the compressed graphs is 0.6 MB, i.e., the space consumed by the model graphs is negligible.

> We conclude that TSFinder is time- and space-efficient enough to document hundreds of classes, e.g., of a third-party library, in reasonable time and with minimal space overhead.

3.6.4 *RQ_4: Comparison with Alternative Approaches*

As the default classification algorithm, we use a SVM with stochastic gradient decent (SGD) and the hinge loss function. We empirically set the learning rate to 0.0001 and the number of WL-iteration h to 3. The following compares this configuration with alternative approaches.

3.6.4.1 *Configuration of the WL Graph Kernel*

To compare field-focused graphs with each other, TSFinder uses the WL graph kernel, which has a parameter h that determines to what extent should it compress node labels. Table 3.4 shows the effect of h on the classification accuracy. The results suggest that $h = 3$ is an appropriate value for h and that small variations of the parameter do not significantly change the accuracy.

TABLE 3.5: Effectiveness of the graph-based TSFinder against the SimpleClassifier classifier.

	Accuracy	
Classifier	TSFinder	SimpleClassifier
SVM[a] (SGD with hinge loss)[b]	94.5%	75.0%
Random forest	94.1%	79.3%
SVM (SMO)[c]	92.5%	70.6%
SVM (SGD with log loss)	92.0%	74.3%
Additive logistic regression	92.8%	74.5%

a Support vector machines
b Stochastic gradient descent
c Sequential minimal optimization

3.6.4.2 *Classification Algorithm*

TSFinder uses a classification algorithm that determines whether a given class vector is likely thread-safe or not (Section 3.4.2.3). We evaluate several other popular algorithms in addition to our default of SVM with stochastic gradient descent and hinge loss. Table 3.5 shows the accuracy of TSFinder with four other classification algorithms, each with the default configuration of hyperparameters provided by Weka. The results show that the accuracy is only slightly influenced by the choice of classification algorithm, as it ranges between 92.0% and 94.5%.

3.6.4.3 *Simple Class-level Features*

We evaluate whether our graph-based view on classes could be replaced by a simpler approach that summarizes class-level features into a vector. The intuition behind this set of features is that as a human, we tend to believe that, for example, a class with high percentage of synchronized methods is probably more likely intended to be thread-safe than a class with fewer synchronized methods. Specifically, we consider the following class-level features:

- Percentage of fields that are volatile.
- Percentage of fields that are public and volatile.

- Percentage of methods that are either synchronized or contain a synchronized block.
- Percentage of methods that are either public and synchronized or public and contain a synchronized block.

Based on a feature vector for each of our 230 classes, we train and evaluate a classifier using the same 10-fold cross validation strategy as above. We call this approach *SimpleClassifier*. The last column in Table 3.5 shows the accuracy obtained by SimpleClassifier using different learning algorithms. All algorithms are used with their default configurations, as provided by Weka. The highest accuracy that SimpleClassifier achieves is 79.3%, using the random forest learning algorithm, which is significantly lower than the accuracy of TSFinder.

> In summary, we find that the choice of classification algorithm has little influence on the accuracy of TSFinder. Comparing the approach with a classifier based on simple, class-level features shows that our graph-based representation of classes yields a significantly more accurate classifier (94.5% versus 79.3%).

3.7 LIMITATIONS

One limitation is that the training classes may not comprehensively cover all possible patterns of thread-safe and thread-unsafe code. As a result, the analysis may not be able to correctly classify a previously unseen class that relies on a completely new way to implement thread safety. We try to address this problem by selecting a diverse set of training classes that are used in various application domains and that cover different concurrency-related implementation patterns, e.g., immutable classes, classes that use synchronized methods, and classes that use synchronization blocks.

Another limitation is that some of the supposedly thread-safe training classes may have subtle concurrency bugs. If such bugs were prevalent, the approach might learn patterns of buggy concurrent code. To mitigate this potential problem, the training set contains well-tested and widely used classes, for which we assume that most of their implementation is correct.

3.8 CONTRIBUTIONS AND CONCLUSIONS

This chapter addresses the understudied problem of inferring concurrency-related documentation. We present TSFinder, an automatic approach to infer whether a class is supposed to be thread-safe or not. Our approach is a novel combination of lightweight static analysis and graph-based classification. We show that our classifier has an accuracy of 94.5% and therefore provides high-confidence documentation, while being efficient enough to scale to hundreds of classes, e.g., in a third-party library.

We envision the long-term impact of this work to be twofold. First, developers of concurrent software can use our approach to decide if and how to use third-party classes. Second, we believe that the technical contribution of this chapter – combining lightweight static analysis and graph-based classification – generalizes to other problems. For example, future work could adapt the idea to other class-level properties, such as immutability, and to other code properties, such as whether a piece of code suffers from a particular kind of bug.

Our evaluation consists of two parts. First, we validate our hypothesis that existing classes lack thread safety documentation by systematically searching all 179,239 classes in the Qualitas corpus [Tem+10]. We find that the vast majority of classes fails to document whether it is thread-safe or not. Second, we evaluate our classifier with 230 training classes that were manually labeled as thread-safe or thread-unsafe. We find that 94.5% of TSFinder's classification decisions are correct. In particular, the precision and recall of identifying thread-safe classes are 94.9% and 94.0%, respectively. On average, adding documentation to a new class takes about 3 seconds. These results show that the approach is accurate enough to significantly improve over guessing whether a class is thread-safe and efficient enough to scale to large sets of classes.

In summary, this chapter makes the following contributions:

- A systematic study of thread safety documentation in real-world Java classes showing the lack of such documentation.
- The first automated classifier to distinguish supposedly thread-safe and thread-unsafe classes, an understudied problem that addresses the lack of thread safety documentation.
- A novel combination of static analysis and graph-based classification that accurately and efficiently predicts the thread safety of a class.

The approach we present in this chapter, TSFinder, supports the main thesis of this dissertation that learning from programs provide an effective means to prevent software bugs. TSFinder infers otherwise missing concurrency specification of object-oriented classes and therefore complements standard bug finding techniques (Chapter 2) by providing a preventive mechanism. In principal, providing such important, but unfortunately missing, documentation helps developers make informed and correct decisions when designing and implementing multi-threaded software.

4

LEARNING TO CROSSCHECK DOCUMENTATION VS. RUNTIME

Traditional static bug detectors, studied in Chapter 2, analyze source code looking for predefined bug patterns. Testing frameworks detect bugs at runtime by checking for known violations, which also have to be prespecified. To define what is a faulty program execution, testing relies on a *test oracle* that distinguishes expected from unexpected behavior. Automatically creating an effective test oracle is difficult, because the correctness of observed behavior depends on what is expected of the software under test. However, the expected behavior is often only informally specified, e.g., in natural language documentation. This chapter presents DocRT, a learning-based approach to *crosscheck documentation against runtime behavior*, effectively providing an alternative solution to one of the challenges for the test oracle problem. More specifically, we focus on exceptional behavior of APIs, i.e., what kind of exception should be thrown, if any, and the pre-condition(s) for such exception(s). The key idea is to exploit natural language information, such as API documentation and names of identifiers and types, which often describes what to expect from a piece of code, but is typically unused during automated testing.

4.1 MOTIVATION

Automated testing is a powerful approach to reveal bugs by exercising a program under test with numerous inputs. A key challenge in automated testing is the oracle problem [Bar+15], i.e., the problem of determining whether a test execution is within the expected behavior or exposes a bug. Without an automated test oracle, a human must reason about test executions, which severely limits the scale at which automated testing can be applied.

A number of test oracles are used in automated testing [Bar+15]. Regression testing provides an oracle by comparing the current behavior of a program to a previous version [LW90; Rot+01]. Differential testing compares two programs that are supposed to behave the same against each other [McK98]. Mined specifications provide a probabilistic oracle that can warn about behavior that deviates from the norm [ABL02; Dal+06; LZ05; PG09; Sho+07; Yan+06]. Some software comes with an implicit specification that may serve as an oracle, e.g., by checking that subclasses follow Liskov's substitutability principle [PG13] or that thread-safe classes have linearizable behavior [PG12]. Another option is to warn about generic signs of misbehavior, such as that program crashes [CS04]. While each of these oracles is effective in some situations, they are likely to miss some bugs even when the misbehavior of the software under test may be obvious to a skilled human observer familiar with the documented behavior of the software.

The reason why automatically creating a test oracle is difficult is that the expected behavior of a software under test is usually only informally specified. Such informal specifications often come in the form of natural language information, e.g., in API documentation or embedded in descriptive method names. While understanding such NL information is relatively easy for a skilled human, e.g., an experienced software developer, automated test oracles typically ignore NL information. The reason is that algorithmically reasoning about NL information is non-trivial, as evidenced by decades of active research in natural language processing (NLP).

As a real-world examples, consider Figure 4.1, which shows a method under test from the widely used Apache Commons Lang project. The API documentation associated with the method `unwrap(String, String)` (Figure 4.1a) describes that the method unwraps a given string (first parameter) using a specified wrapping token (second parameter). Moreover, the documentation also gives several examples of invocations of this method and the expected output. However, as shown by the test case in Figures 4.1b, the method throws a `StringIndexOutOfBoundsException` when the length of the string to be unwrapped equals the length of the wrapping token (Figure 4.1c). This behavior is wrong according to the documentation, and indeed the fix of this bug changes the logic of the source code to safely handle this corner case.[1] While the fact that this observed behavior is wrong is obvious for a skilled human, automated test oracles typically fail to see a problem, because they do not "understand" the NL documentation.

1 https://issues.apache.org/jira/browse/LANG-1475

```
public static String unwrap(String str, String wrapToken)

Unwraps a given string from another string.
...
StringUtils.unwrap("AABabcBAA", "AA") = "BabcB"
...
StringUtils.unwrap("#A", "#") = "#A"
...
Parameters:
str - the String to be unwrapped, can be null
wrapToken - the String used to unwrap
Returns:
unwrapped String or the original string if it is not quoted properly with the
      wrapToken
```

(a) Documentation of method under test.

```
org.apache.commons.lang3.StringUtils.unwrap("a", "a");
```

(b) Test case for the method under test.

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String
      index out of range: -1
  at java.lang.String.substring(String.java:1967)
  at org.apache.commons.lang3.StringUtils.unwrap (StringUtils.java:9345)
  at lang3_9.RegressionTest0.test05(RegressionTest0.java:50)
  at lang3_9.Main.main(Main.java:12)
```

(c) Runtime behavior of the method under test

FIGURE 4.1: The method under test `unwrap(String,String)` from the Apache Commons Lang library with associated NL information that hints at the fact that the behavior of the shown test is unexpected.

This chapter presents DocRT, which addresses the test oracle problem through a learning-based approach. In particular, we focus on the problem of exceptional behavior of APIs. The key idea is to predict whether the observed behavior of a method under test conforms to the NL information associated with this method. The core of DocRT is a learned model that takes two inputs: (1) different kinds of NL information associated with a method under test, such as its API documentation, the name of the method, and the name of its parameters; (2) a summary of the states before and after a call of the method, e.g., the value of parameters, the return value, or an exception the method may throw. Given these two inputs, the model predicts whether the observed behavior conforms to the NL information.

If the observed behavior is likely to contradict the NL information, then DocRT reports a warning, effectively providing an NL-based, automated test oracle.

In principle, a wide range of approaches could address the problem of predicting whether observed execution behavior and NL information match. We here present a learning-based approach, and more specifically a deep learning model, because such models have shown tremendous success in NLP in the past few years [Boj+17; Col+11; Mik+13]. Neural models can successfully identify a wide range of patterns in a given dataset and so "understand" the meaning of specific NL information. As software, even when developed by different developers and in different application domains, tends to be repetitive [Hin+12], it provides an excellent target for learning recurring patterns.

A key challenge in any supervised learning approach is obtaining a suitable dataset for training. To train the DocRT model, we gather hundreds of thousands of method executions and their associated NL information. The data is the result of combining an existing unit test generator [Pac+07], a mechanism to extract program states at runtime, and a mechanism to extract NL information from code. The resulting data provides examples of likely correct pairs of behavior and NL information. To also obtain negative training examples, DocRT mutates the examples by mimicking potential mistakes, such as undocumented exceptional behavior or wrongly documented behavior.

4.2 APPROACH

4.2.1 *Problem Statement*

The goal of this work is to predict whether the observed runtime behavior of a method under test (MUT) corresponds to its natural language documentation.

Definition 4.2.1 (Problem) *Let M_{NL} be the natural language description of method M and let $M_{runtime}$ be the observed runtime behavior of a single invocation of M. The problem is to probabilistically predict whether $M_{runtime}$ and M_{NL} match through a a binary classifier $C : (M_{NL}, M_{runtime}) \rightarrow [0, 1]$. When using C as an oracle, a prediction of 0 means certainly not buggy and 1 means certainly buggy.*

A mismatch between $M_{runtime}$ and M_{NL} can be due to two reasons:

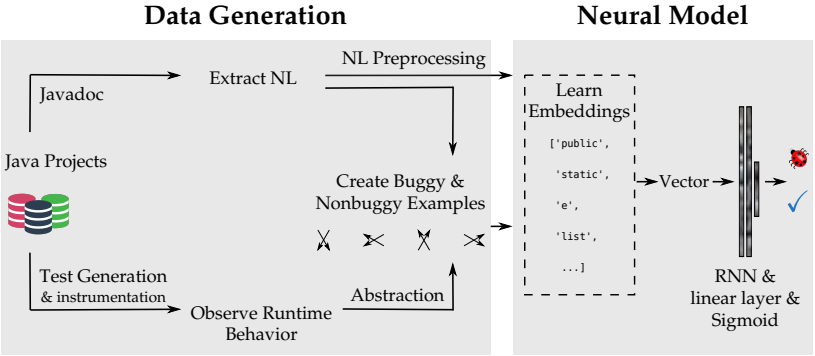


FIGURE 4.2: Overview of DocRT: Learning-based crosschecking of documentation and runtime.

- The NL description is wrong or incomplete with respect to the method behavior, i.e., the observed behavior is actually intended but documented poorly. In this case, the method documentation should be fixed.
- The method behavior is wrong or unexpected with respect to the documented behavior, i.e., the observed behavior is wrong. In this case, the method implementation should be fixed.

4.2.2 *Overview*

Figure 4.2 provides an overview of the DocRT approach for learning of implicit test oracles from NL and observed program behavior. DocRT requires a natural language description of the behavior of a MUT and a manifestation of that behavior in terms of runtime execution. Therefore, DocRT gathers natural language information about the MUT from several sources, as well as several traces of the MUT execution at runtime.

The first step of our approach collects several hundreds of Java projects from Apache Maven[2], which serve as training and evaluation data (Section 4.2.3). Second, the approach extracts NL information describing the methods in these projects. The NL information comes from different sources, such as the method names, parameter names, and documentation (Section 4.2.4). Third, using the downloaded projects and an existing automatic test generator [Pac+07], the approach collects thousands of execution traces

2 http://maven.apache.org

of methods exposed via public APIs (Section 4.2.5). The fourth steps is to represent the NL information and the execution traces into a format suitable for learning, which DocRT addresses by learning word embeddings and by abstracting execution states into vectors (Section 4.2.7). Finally, the approach trains a binary classifier in the form of a neural network (Section 4.2.7.3).

4.2.3 *Collecting Projects from Maven*

To obtain both natural language descriptions and runtime execution values for thousands of methods, we use projects from Maven, which provides a uniform interface to the different artifacts required by DocRT. We first download the entire Maven index and search it for artifacts that are packaged as jar files and that have their source code available. The reason for choosing jar packaging is that it is one of the most widely used packaging formats, it is compatible with the automatic test generator input format, and it is also a suitable format for uniformly collecting the method documentation, as we explain later in Section 4.2.4.

From the filtered Maven Index, we randomly sample 5,000 projects. For each project, we use the Maven command line tool to download three kinds of artifacts for each project: (i) The project sources jar (sources); (ii) The project binary jar (binary); (iii) All jars required to compile the project sources (compile-path).

4.2.4 *Gathering NL Information*

To automatically build a test oracle for a target MUT, we need two kind of inputs, where the first is a description of the expected behavior of the method. There are various sources that could provide useful information about the functionality of an API, such as the project description, the API documentation, FAQs about the API, and the names of API elements, e.g., classes, methods, and parameters.

In this work, we use API documentation along with method and parameters names and types. Our intuition is that, according to good software engineering practices, developers are encouraged to use descriptive names for the different API elements. Moreover, method-level documentations serve as an informal contract between the API developer and the API user. Method documentation describes what is the expected input of the method and the expected output. Besides, documentation of methods should also

specify the method behavior under faulty invocations, e.g., the kinds of exceptions thrown when presented with unexpected argument.

Since we use Java projects, we collected all the documentation information as well as method and parameter names and types from the Javadoc of each method. Javadoc comments are structured class-level and method-level descriptions of classes and methods, which include various tags, such as `@param`, `@return`, to describe the method parameters and return values, respectively. DocRT collects the following information for each MUT.

4.2.4.1 *Method Name and Description*

DocRT extracts the fully qualified method name M starting from the top package. For the example in Figure 4.1, the approach extracts the method name starting from `org` up until the method identifier `unwrap`. Moreover, the main body of the method description $desc(M)$ is also extracted from the documentation of M. This is the main paragraph describing the method functionality but none of the tags, such as `@parameter`, `returns`, and `@throws`.

4.2.4.2 *Return Type and Description*

If the method return type is not `void`, DocRT also extracts the fully qualified return type M^{τ}_{ret} of the the method and the NL comment $desc(M_{ret})$ describing the return value, which is specified by the Javadoc tag `@return`.

4.2.4.3 *Parameter Names, Types, and Descriptions*

For a method M with parameters $p_1, p_2, \ldots, p_n$, the approach extracts their corresponding fully qualified types $p^{\tau}_1, p^{\tau}_2, \ldots, p^{\tau}_n$, and the NL descriptions $desc(p_1), desc(p_2), \ldots, desc(p_n)$ of each parameter, as given by the Javadoc tags `@param`. DocRT assembles the following list of information:

$$M_{params} = [(p_1, p^{\tau}_1, desc(p_1)), (p_2, p^{\tau}_2, desc(p_2)), \ldots, (p_n, p^{\tau}_n, desc(p_n))]$$

If the method accepts no parameters, the list M_{params} is empty.

4.2.4.4 *Type and Description of Thrown Exceptions*

When the method defines how it it handles erroneous behavior through runtime exceptions, the Javadoc provides API developers with the `@throws` tag. It allows the developer to specify the type $throws^{\tau}$ of the thrown

exception and a description $desc(throws^{\tau})$ of the condition under which this exception is thrown. Since a method could throw multiple kinds of exceptions for different faulty behaviors, the `@throws` tag can be used multiple times in the Javadoc of a method. Hence, DocRT extracts the following set of tuples that summarize the documentation about thrown exceptions:

$$M_{throws} = \{(throws_1^{\tau}, desc(throws_1^{\tau})), (throws_2^{\tau}, desc(throws_2^{\tau})), \ldots, (throws_n^{\tau}, desc(throws_n^{\tau}))\}$$

In summary, the NL information extracted for a method M is:

$$M_{NL} = \left(M, desc(M), M_{ret}^{\tau}, desc(M_{ret}), M_{params}, M_{throws}\right)$$

4.2.5 *Capturing Runtime Behavior*

The second kind of input required for learning the DocRT test oracle is a summary of the runtime behavior of a method. Capturing the runtime behavior of a MUT is non-trivial for two reasons:

- *What to capture?* We need to choose what information exactly to capture. For instance, one could represent the method behavior by recording the sequence of method calls a the MUT performs, the memory values it reads or writes, or the control flow decisions it takes. Another approach would be to record a memory snapshot before and after invoking the MUT.
- *How to abstract the runtime behavior?* To feed the captured runtime behavior into a machine learning model, the information must be abstracted in a suitable way.

4.2.5.1 *Extracting Pre- and Post-States of MUT Calls*

We address these challenges by capturing four kinds of information about the runtime behavior of a MUT:

(a) The base object, if any, before and after the call;

(b) The values of the method arguments, if any, before and after the call;

(c) The return value of the method, if any;

(d) The exceptional behavior, in case the method throws an exception.

The rationale for focusing on these four pieces of information is that their roughly correspond to pre- and post-conditions of the MUT, which is what informal API documentation also tries to describe. We now explain in more detail how DocRT captures the runtime behavior.

BASE OBJECT If M is an instance method, then DocRT extracts the type $base^{\tau}$ and the state of the base object, denoted M_{base}. Since an invocation of a MUT instance could potentially change the state of its base object, we extract the pair of pre- and post-states of the base object $(base^{pre}, base^{post})$. If the M is a static method, then M_{base} is empty.

METHOD ARGUMENTS If M accepts one ore more arguments $arg_1, arg_2, \ldots, arg_n$, DocRT captures the list of the values of these arguments, again in pairs of pre- and post-states per argument:

$$M_{args} = \left[(arg_1^{pre}, arg_1^{post}), (arg_2^{pre}, arg_2^{post}), \ldots, (arg_n^{pre}, arg_n^{post})\right]$$

If the method accepts no arguments, then this list is empty.

METHOD RETURN VALUE If M returns a value, then DocRT captures the returned value M_{ret}. Otherwise, the return value is none.

EXCEPTIONAL BEHAVIOR When calling M triggers an exception thrown back to the caller, the approach captures the type of exception thrown M_{thrown}^{τ}.

ABSTRACTING THE STATE OF OBJECTS The state of objects can be arbitrarily complex. DocRT abstracts the captured states based on a series of abstraction functions that encode special values, such as `null`, that capture abstract properties of specific types of values, such as the size of a collection, and that summarize the fields of an object into key-value pairs. To extract the pre- and post-states, the approach instruments the code that invokes the MUT via AST-based transformations. Section 4.3 describes how we implement the extraction and abstraction of pre- and post-states.

In summary, the runtime behavior DocRT captures is given by:

$$M_{runtime} = \left(base^{pre}, base^{post}, M_{args}, M_{ret}, M_{thrown}^{\tau}\right)$$

4.2.5.2 *Exercising the Methods Under Test*

To capture runtime behavior, as described above, DocRT relies in executions of the MUTs. One option would be to use test cases written by the developers of each project. However, there are two problems with this approach. First, it assumes that there are test cases for a large number of projects (5,000), but not all projects provide an extensive test suite. Second, the quality and number of test cases, when they exist, could differ heavily between different projects.

Instead of relying on developer-written tests, DocRT builds on an automated test generator. We opted to use Randoop, a state-of-the-art automatic test generator for Java applications [Pac+07]. We set Randoop to test public methods only, with time budget of three minutes per project. Overall, this setup yields hundreds of thousands of test executions (Section 4.4.1), providing DocRT with plenty of executions to capture the runtime behavior from.

4.2.6 *Generating Buggy Examples*

Since we formulate test oracle problem as a binary classification problem (Definition 4.2.1) and since we aim at training a supervised model, the approach requires a large number of examples of both matching and non-matching pairs $(M_{NL}, M_{runtime})$. One option could be to manually inspect method calls and the corresponding documentation, which however, does not scale to the number of training examples needed to obtain an effective model. Instead, we assume that the pairs of documentation and runtime behavior we have collected from the real-world Java projects are mostly non-buggy, and create buggy examples by mutating these real-world pairs. Creating buggy data points through mutations of real-world code has been successfully used for learning static bug detectors [PS18]. We here explore whether this idea can be adapted to learning test oracles.

There are many possible ways of mutating a given, supposedly correct $(M_{NL}, M_{runtime})$ pair into a likely incorrect pair. DocRT focuses on a set of four kinds of mutations, inspired by real-world bugs where the documentation and the actual behavior mismatch. The mutations focus on exceptional behavior and its documentation, because unexpected exceptions may have a severe impact on API clients. The overall DocRT approach could be easily extended with other kinds of mutations that model other kinds of mismatches between documentation and runtime behavior.

4.2.6.1 *Raise Random Exception*

This mutation replaces a non-exceptional post state of the MUT by a randomly selected type of exception, imitating the situation where the invocation causes unexpected exceptional behavior. For a positive example

$$x^+ = \left(M_{NL}, base^{pre}, base^{post}, M_{args}, M_{ret}\right)$$

applying the mutation yields the following negative example:

$$x^- = \left(M_{NL}, base^{pre}, M_{args}, sample(M^{\tau}_{thrown})\right)$$

where $sample(M^{\tau}_{thrown})$ denotes a random exception type sampled from the set of thrown exceptions in the dataset. The sampling follows the distribution of exceptions in the dataset, to create a realistic negative example.

4.2.6.2 *Remove Thrown Exception*

This mutation replaces the post-state where an exception is thrown by a randomly sampled valid post-state of the another invocation of the same method.

$$x^+ = \left(M_{NL}, base^{pre}, M_{args}, M^{\tau}_{thrown}\right)$$

yields

$$x^- = \left(M_{NL}, base^{pre}, sample_M(base^{post}, M_{args}, M_{ret})\right)$$

where $sample_M$ randomly samples a non-exceptional post-state among all invocations of M.

4.2.6.3 *Replace Thrown Exception by Random Exception*

If the method post-state already raises an exception, this mutation replaces this exception by another randomly sampled type of exception. Given the positive example

$$x^+ = \left(M_{NL}, base^{pre}, M_{args}, M^{\tau}_{thrown}\right)$$

the mutation yields

$$x^- = \left(M_{NL}, base^{pre}, M_{args}, sample(M^{\tau}_{thrown})\right)$$

where $sample(M^{\tau}_{thrown}) \neq M^{\tau}_{thrown}$.

4.2.6.4 *Remove Raised Exception from Documentation*

If the method post-state already raises an exception, and this exception is properly documented in the API documentation, this mutation removes the documentation of this thrown exception. Given

$$x^{+} = \left(\ldots, throws_i^{\tau}, desc(throws_i^{\tau}), \ldots, M_{thrown}^{\tau}\right)$$

where $throws_i^{\tau} = M_{thrown}^{\tau}$, the mutation yields

$$x^{-} = \left(\ldots, throws_{i-1}^{\tau}, desc(throws_{i-1}^{\tau}), \right.$$
$$\left. throws_{i+1}^{\tau}, desc(throws_{i+1}^{\tau}), \ldots, M_{thrown}^{\tau}\right)$$

4.2.7 *Learning the DocRT Model*

The final step of the approach is to train a machine learning model that classifies a given pair $(M_{NL}, M_{runtime})$ as correct or buggy. To this end, the approach converts the different NL and runtime information into numerical vectors suitable for a neural network-based model. We choose a neural model, instead of of a more traditional, feature-based model, because neural models have been shown to be very effective in reasoning about NL information, both in natural language processing [Col+11] and in program analysis [MPP19; PS18].

4.2.7.1 *Embeddings*

DocRT embeds the tokens from the NL and runtime information into a vector space. An embedding maps an entity, a token in our case, to a many-dimensional, real-valued vector space, while preserving semantic similarities between the entities. We pre-train an embedding model on API documentation using a state-of-the-art embeddings techniques based on subword n-grams, which is able to handle out-of-vocabulary tokens [Boj+17].

We train the embedding model on documentation and then use it for both the NL and the runtime information for three reasons:

- Documentation is a natural source for the description of the method behavior.
- Documentation also includes informal descriptions of pre-, and post-state of methods.

- Documentation sometimes includes values, e.g, of arguments and the return values of methods.

The embedding model is trained on the documentation of all the 207,455 methods DocRT managed to extract documentation for.

4.2.7.2 *Vector for Learning*

DocRT assembles all gathered NL information about the MUT and the information captured from a single execution of the MUT into an input vector for the learned model.

Definition 4.2.2 (Input vectors) *The vector DocRT uses to learn test oracles is composed of NL components extracted from the MUT documentation and the observed execution values related to the MUT.*

$$
\begin{aligned}
vec = \big[& M, desc(M), \\
& p_1^\tau, p_1, desc(p_1), arg_1^{pre}, arg_1^{post}, \ldots, \\
& p_n^\tau, p_n, desc(p_n), arg_n^{pre}, arg_n^{post}, \\
& M_{ret}^\tau, desc(M_{ret}), M_{ret}, \\
& throws_1^\tau, desc(throws_1), \ldots, \\
& throws_m^\tau, desc(throws_m), thrown^\tau, \\
& base^\tau, base^{pre}, base^{post} \big]
\end{aligned}
$$

In this vector, several components are concatenated in a such a way that related pieces of information are next to each other. For example, each method argument value is concatenated next to the type, name, and description of the corresponding parameter. This arrangement helps the model to spot inconsistencies between the NL and runtime information associated with a specific aspect of the MUT.

4.2.7.3 *Neural Model*

Given input vector labeled as correct and incorrect, DocRT trains a neural classifier that learns to distinguish these two kinds of inputs. The classifier is a bi-directional, long short-term memory, recurrent neural network (bi-LSTM RNN) with two hidden layers each of size 100. The RNN summarizes the given input vector into a hidden state, which is then fed through a fully connected layer. Finally, the model outputs an output vector that indicates the probability that the given input vector describes non-matching NL and

runtime information. The output layer uses the sigmoid activation function and a dropout of 0.4. The model is trained with stochastic gradient descent using the AdamW optimizer (with weight decay) [Zha18], based on binary cross entropy (BCE) for computing the loss. During training, the expected output is zero for all positive examples x^+ and one for all negative examples x^-.

4.3 IMPLEMENTATION

We use the Apache Maven Indexer API[3] to download the entire Maven Index and Lucene[4] to search the index for artifacts matching our criteria (Section 4.2.3). To extract documentation, we first build the HTML documentation files for each public class and its public methods using the Javadoc tool, and then extract the relevant parts of the documentation using or own HTML parser, which is written in Python based on the Beautiful Soup library[5]. The reason for generating the HTML documentation from the source code instead of using the pre-packaged documentation jars available via Maven are differences in the HTML trees due to different Javadoc or HTML versions. We ignore methods that have neither a description of their functionality nor a description of their return value.

To exercise the methods in the dataset, we use Randoop [Pac+07]. DocRT instruments the Randoop-generated test cases to capture the runtime values of pre- and post-states of method arguments and base objects, return values, and exceptional behavior. The instrumentation is implemented as AST-based transformations based on the JavaParser[6].

To abstract the runtime states of objects, DocRT serializes all primitive values of method arguments and return values. To capture non-primitive values, such as the state of the base object and any non-primitive arguments or return values, we use a set of rules: 1. Use Apache BeanUtils[7] to get the object state in terms of key-value pairs using all available object getters. 2. If (1) fails, use Google's Gson serialization library[8] to obtain a property-based string representation of the object using public fields only. 3. If (2) fails, use the Jakarta JSON binding library[9] to serialize the object. 4. If (3) fails,

3 http://maven.apache.org/maven-indexer
4 https://lucene.apache.org
5 https://www.crummy.com/software/BeautifulSoup
6 https://javaparser.org
7 http://commons.apache.org/proper/commons-beanutils
8 https://github.com/google/gson
9 http://json-b.net

flag the object unserializable and abort. The different serialization scenarios all yield a key-value map of the underlying object state. The intuition of using property-based and field-based serialization is that either way, we get a peek into the object state. Finally, in addition to the above, if the object implements the `Iterable` interface, we query its size and prepend a size key at the beginning of the serialized object.

To prepare the documentation and serialized runtime values for learning, we apply standard preprocessing by removing stop words, punctuation from key-value maps of runtime values, and lemmatization using NLTK [10].

To train embeddings for documentation and runtime value tokens, we train a FastText embedding model [Boj+17], as implemented in the gensim library[11]. The neural classifier of DocRT is built on top of PyTorch[12].

4.4 EVALUATION

Our evaluation applies DocRT to methods from 5,000 open-source Java projects. We address the following research questions:

- RQ_1: How effective is the approach at distinguishing correct from incorrect behavior?
- RQ_2: How effective is the approach at detecting real-world bugs?
- RQ_3: How efficient is the approach?

4.4.1 *Experimental Setup*

We download 5,000 Java projects from Maven. DocRT gathers documentation for 207,455 public Java methods from those projects. We train the FastText word embedding model [Boj+17] on the entire set of method documentation for 20 epochs, with window size of 5, a minimum word frequency of 1, and an embedding size of 50. Randoop managed to generate 146,397 tests for 25,076 methods in the dataset of methods with documentation. After creating the buggy examples using the mutations in Section 4.2.6, there are 292,782 pairs $(M_{NL}, M_{runtime})$ of method documentation and runtime behavior. The dataset is balanced, i.e., it contains roughly the same number of buggy and non-buggy examples.

10 https://www.nltk.org
11 https://radimrehurek.com/gensim
12 https://pytorch.org

Before running the experiments, we split the dataset into three parts: for training (70%), validation (15%), and testing (15%). The validation data, but not the test data, is used to tune the hyper-parameters of the neural classifier model. Unless otherwise mentioned, all reported results are on the test set.

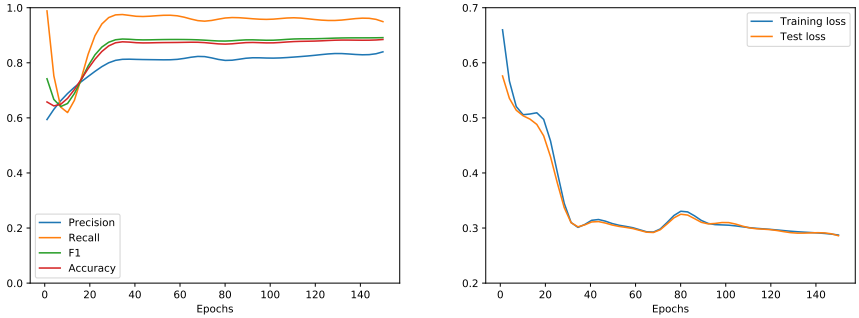
We experiment with different values for the hyper-parameter of the neural classifier of DocRT. For the hidden layers, we use one, two, and three layers with sizes of either 100 and 200. For dropout, we experiment with 0.2, 0.3, and 0.4. For the learning rate, we try 0.001, 0.0005, and 0.0001. For the batch size, we experiment with 512, 1,024, and 2,048. By exploring different combinations of these hyper-parameters and evaluating the resulting models on the validation data, we find the following configuration to provide the best overall accuracy: Two hidden layers of size 100, dropout of 0.3, batch size of 1,024, a decaying learning of 0.0001, and 50 epochs.

All experiments are done on a machine with one NVIDIA Tesla V100 GPU, 48 Intel Xeon CPU cores with 2.2Ghz, and 250 GB RAM, which is running Ubuntu Linux 18.04.

4.4.2 *RQ₁: Effectiveness of Learned Model*

The learned model that predicts whether a method execution is consistent with the NL information is the core of the DocRT approach. We evaluate the effectiveness of this model in distinguishing correct from incorrect method executions. To this end, we train the model with the training data, then apply the trained model to the test data, and compare the predictions to the ground truth. Based on this comparison, we compute the following metrics. $P_{\checkmark}$ and $P_{\times}$ are the sets of calls the model predicts to be correct and incorrect, respectively. $G_{\checkmark}$ and $G_{\times}$ are the sets of calls that are labeled as correct and incorrect, respectively, in the ground truth.

- Accuracy, which is the percentage of predictions that match the ground truth among all predictions: $\frac{|(P_{\checkmark} \cap G_{\checkmark}) \cup (P_{\times} \cap G_{\times})|}{|P_{\checkmark} \cup P_{\times}|}$
- Precision, which indicates how often the model is right when predicting that a call is incorrect: $\frac{|P_{\times} \cap G_{\times}|}{|P_{\times}|}$
- Recall, which indicates how many of all incorrect the model finds: $\frac{|P_{\times} \cap G_{\times}|}{|G_{\times}|}$
- F1-score, which is the harmonic mean of precision and recall.



(a) Effectiveness of the classifier.

(b) BCE loss of the classifier.

FIGURE 4.3: Training DocRT over epochs.

Figure 4.3a shows the effectiveness of the learned model depending on the number of training epochs. As is common with deep learning models, the effectiveness fluctuates initially and then stabilizes after some number of epochs. Once stabilized, the model achieves an accuracy of 87%, a precision of 81%, a recall of 97%, and an F1-score of 88%. These results show that the classifier is highly effective at distinguishing pairs of NL information and runtime behavior that match from those that mismatch.

To illustrate the learning process, Figure 4.3b shows the overall loss of the model in the validation and on the training data. Both losses are roughly the same, indicating that the model does not simply overfit to the training data but learns to generalize to other examples.

4.4.3 *RQ_2: Detecting Real-world Bugs*

The ultimate goal of a test oracle is to detect bugs. We address the question of how effective DocRT is at detecting real-world bugs in two ways. First, we apply the approach to a set of *previously known bugs* that are related to an inconsistency between the documented and actual behavior. We focus on bugs described in issue trackers and that were fixed by the developers. This experiment is useful to validate that the approach finds problems relevant to developers and gives an idea how many of a set of such bugs the approach can find. Second, we inspect a subset of those calls in the test dataset that the DocRT model classifies as incorrect, but that correspond to the actual behavior and documentation of the tested code. While in RQ_1, we consider all of these calls as correct, we suspect that some fraction of

them is actually incorrect. Since determining the expected behavior and comparing it to the actual behavior is a time-consuming process, we inspect only a subset of 50 such calls. This experiment shows how effective DocRT is at detecting *previously unknown bugs*.

4.4.3.1 *Previously Known Bugs*

Table 4.1 shows the set of previously known bugs that we apply DocRT to. We select these bugs by searching through issue trackers for bugs where the behavior of a method clearly diverges from the API documentation. In total, we gather 24 such bugs from ten bug reports. Instead of using Randoop for generating tests, we opted to use the tests associated with the bug reports as these tests are known to reveal the buggy behavior of the methods. Using those tests, we instruct DocRT to instrument those test cases to gather the runtime values and to extract the documentation of the respective methods. For six out of the 24 methods, DocRT fails to extract the pre- and post-runtime values. The reason is that when using reflection to serialize arguments or base objects, exceptions were thrown and the instrumentation hence fails to capture the values.

The table lists the remaining 18 bugs and whether the approach successfully identifies a call to the buggy method as buggy. The results show that DocRT successfully detects all 18 bugs. Although all these bugs are mismatches between the documentation of a MUT and the corresponding runtime behavior, the developers took different paths to fix them. For example, the first bug in Table 4.1 was fixed by updating the documentation to reflect the thrown exception, while bugs 2 to 4 were fixed by changing the source code to avoid the thrown exception. Yet another approach was used to fix bugs 4 and 5, where the developers deprecated the constructor that led the object to be in a state where those method would behave erroneously.

Table 4.1: Known bugs and whether DocRT finds them.

	BugId	Method	Description	Detected
		Apache Commons Lang		
1	LANG-1426	StringUtils.truncate(String,int,int)	Undocumented IllegalArgumentException	✓
2	LANG-1475	StringUtils.unwrap(String,String)	Unexpected IndexOutOfBounds in corner case	✓
3	LANG-1406	StringUtils.removeIgnoreCase(String,String)	Unexpected IndexOutOfBounds in corner case	✓
4	LANG-1453	StringUtils.replaceIgnoreCase(String,String,String)	Unexpected IndexOutOfBounds in corner case	✓
		Apache Commons Math		
5	MATH-1116	optim.nonlinear.vector.MultivariateVectorOptimizer.getWeight()	Undocumented NullPointerException	✓
6	MATH-1116	optim.nonlinear.vector.MultivariateVectorOptimizer.getTarget()	Undocumented NullPointerException	✓
7	MATH-1116	optim.nonlinear.scalar.noderiv.AbstractSimplex.getPoint(int)	Undocumented NullPointerException	✓
8	MATH-1116	optim.nonlinear.scalar.noderiv.SimplexOptimizer.optimize(optim.OptimizationData...)	Undocumented NullPointerException	✓

(Continued on next page)

TABLE 4.1: Known bugs and whether DocRT finds them.

	BugId	Method	Description	Detected
9	MATH-1116	random.ValueServer.resetReplayFile()	Undocumented NullPointerException	✓
10	MATH-1116	random.EmpiricalDistribution. getGeneratorUpperBounds()	Undocumented NullPointerException	✓
11	MATH-1116	stat.correlation.PearsonsCorrelation. getCorrelationStandardErrors()	Undocumented NullPointerException	✓
12	MATH-1116	stat.regression. AbstractMultipleLinearRegression. estimateErrorVariance()	Undocumented NullPointerException	✓
13	MATH-1116	stat.regression. OLSMultipleLinearRegression. calculateHat()	Undocumented NullPointerException	✓
14	MATH-1224	ode.AbstractIntegrator. computeDerivatives(double,double[],double[])	Undocumented NullPointerException	✓
15	MATH-1224	stat.correlation.SpearmansCorrelation. getCorrelationMatrix()	Undocumented NullPointerException	✓
16	MATH-1401	stat.interval.IntervalUtils. getClopperPearsonInterval(int,int,double)	Unexpected custom math exception in corner case	✓

(Continued on next page)

Table 4.1: Known bugs and whether DocRT finds them.

	BugId	Method	Description	Detected
		Apache Commons Collections		
17	COLL.-701	list.SetUniqueList.add(Object)	Infinite recursion when adding itself	✓
18	COLL.-727	iterators.CollatingIterator.setIterator(int,Iterator)	Wrong equality in @throws condition	✓

TABLE 4.2: Summary of previously unknown bugs detected by DocRT.

Analyzed method calls	21,592
Calls where DocRT reports a warning	4,829
Inspected calls with warning	50
True positives	45
Undocumented exception	39
Wrong type of exception documented	1
False positives	5
Indirect description of the exceptional behavior	4
Exceptional behavior is fully documented	1

4.4.3.2 *Previously Unknown Bugs*

To evaluate if and how precisely DocRT detects previously unknown bugs, we manually inspect a subset of the supposedly correct calls in the test data set that the approach classifies as incorrect. Table 4.2 summarizes the results of this experiment. The number of real-world method calls, i.e., not created by mutations, in the held-out test set is 21,592. When considering every prediction above 0.5 as a warning, then DocRT classifies 4,829 out of those 21,592 calls as buggy. In practice, we envision developers to inspect the methods with the highest predicted probability first.

We sample 50 of the top predictions for manual inspection. For each method, we carefully inspect the documentation of the MUT, the generated test case, and the exposed runtime behavior. If the described and the actual behavior clearly diverge, then we consider the warning as a true positive. Otherwise, we consider the warning to be a false positive.

Overall, we find that 45 of the inspected 50 warnings are true positives, i.e., method calls exposing behavior that does not match the documentation. Figure 4.4 shows two of the true positives for illustration. Figure 4.4a shows a bug detected by DocRT in the Apache Fluo project, where the thrown exception does not match the type of exception specified in the method documentation. Figure 4.4b shows another bug, detected in the Apache ActiveMQ project. In this bug, the documented exception is correct in the sense that `java.io.DataOuput` could potentially throw a `java.io.Exception`. However, the documentation does not mention the

risk of a `NullPointerException` being thrown, i.e., the example is an undocumented exception. Both bugs have already been fixed in response to our reports of the problems to the developers.

The results of the manual inspection show that DocRT successfully detects previously unknown documentation-related bugs. The fact that most of inspected warnings are true positives also suggests that the precision of DocRT is likely to be higher than the precision measured in RQ_1, which assumes that the analyzed MUTs are bug-free.

4.4.4 *RQ_3: Efficiency*

Our evaluation the efficiency of DocRT focuses on the time required by the neural model, because this is the most time-consuming part of the approach. Training DocRT on a dataset of 204,970 examples takes on average 2.5 minutes per epoch. In total, it takes around 5.5 hours to train DocRT for 50 epochs. To classify a single pair of NL information and runtime behavior, DocRT takes less than a second.

In addition to the time spent on training and using the neural model, the DocRT approach also requires time for extracting NL information from API documentation, for generating tests, and for capturing runtime information during test execution. The test generation time depends on the test generator used, and improving it is beyond the scope of this work. The other time costs are negligible compared to the neural model.

4.5 CONTRIBUTIONS AND CONCLUSIONS

Automated testing is promising but requires test oracles to distinguish correct from incorrect executions. This chapter tackles the old test oracle problem in a new way: By learning a model that predicts whether the observed runtime behavior is in line with the NL documentation. The approach first gathers pairs of runtime behavior from executions driven by an automated test generator and NL information associated with the tested methods, and then trains a neural model to classify these pairs as correct or buggy. To create buggy examples for training, we use a set of mutation operators that mimic common documentation-related bugs. One interesting insight from this work is that training with artificially created negative examples eventually yields a model able to find real-world bugs.

Documentation:

```
public byte byteAt(int i)

Gets a byte within this sequence of bytes

Parameters:
i - index into sequence
Returns
byte
Throws:
IllegalArgumentException - if i is out of range
```

Test case:

```
1 org.apache.fluo.api.data.Bytes bytes = new
      org.apache.fluo.api.data.Bytes();
2 bytes.byteAt(-1);
```

Observed runtime behavior:

```
Exception in thread "main"
      java.lang.IndexOutOfBoundsException: i < 0, -1
at org.apache.fluo.api.data.Bytes.byteAt(Bytes.java:106)
at fluo_1_2.RegressionTest0.test01(RegressionTest0.java:14)
at fluo_1_2.Main.main(Main.java:8)
```

(a) Bug detected in Apache Fluo project. The actual exception type thrown by the method `byteAt` does not match the type of exception documented in the `@throw` tag.

Documentation:

```
public void marshal(java.lang.Object command,
      java.io.DataOutput out) throws java.io.IOException

Specified by:
marshal in interface org.apache.activemq.wireformat.WireFormat
Throws:
java.io.IOException
```

Test case:

```
1 org.apache.activemq.transport.xstream.XStreamWireFormat
      format = new org.apache.activemq.transport.xstream.
      XStreamWireFormat();
2 java.io.DataOutput out = null;
3 format.marshal("", out);
```

Observed runtime behavior:

```
Exception in thread "main" java.lang.NullPointerException
at org.apache.activemq.transport.util.TextWireFormat.
      marshal(TextWireFormat.java:47)
at activemq_5_15_11.RegressionTest0.
      test01(RegressionTest0.java:14)
at activemq_5_15_11.Main.main(Main.java:8)
```

(b) Bug detected in Apache ActiveMQ project. The `marshal` method throws a `NullPointerException` whereas documentation mentions `IOException` only.

FIGURE 4.4: Examples of previously unknown bugs detected by DocRT.

Applying DocRT to 25,076 Java methods from a variety of application domains shows the effectiveness of the approach. The learned oracle model achieves an accuracy of 87%, a precision of 81%, and a recall of 97%. Specifically, our results show that the approach finds 18 previously known and 45 previously unknown bugs in popular Java software—bugs that are caused by inconsistencies between behavior and documentation. All bugs are related to exceptional behavior and how (if at all) it is documented, as this is the focus of our training data.

In summary, this chapter contributes the following:

- *Problem*: We are the first to formulate the test oracle problem as the problem of predicting whether observed runtime behavior and associated NL information match.
- *Approach*: We present DocRT, which addresses the above problem with a learning-based approach that reasons about NL information and runtime behavior using a neural classification model.
- *Effectiveness*: We show that DocRT identifies mismatches between behavior and documentation with an accuracy of 87%, enabling the approach to find 18 known and 45 previously unknown bugs in widely used Java code.

This chapter supports the central thesis of this dissertation by presenting DocRT, a novel learning-based approach to crosscheck API documentation and runtime behavior for potential inconsistencies. Our experience with the reported bugs show that not only is DocRT effective, but it also detects bugs in both the documentation and the program runtime behavior. Our insight here is that documentation provides invaluable and readily available information which can help in detecting and preventing bugs, but is often overlooked by traditional bug detection techniques, whether static or dynamic.

5

FROM DOCUMENTATION TO SUBTYPE CHECKING

JSON (JavaScript Object Notation) is a popular data format used pervasively in web APIs, cloud computing, NoSQL databases, and increasingly also machine learning. One of the main advantages of JSON is that it is both human- and machine-readable. To ensure that JSON data is compatible with an application, one can define a JSON Schema and use a validator to check data against the schema. JSON schemas serve two purposes: (i) They provide an effective means to automatically validate JSON data ensuring it conforms to a predefined specification. (ii) They serve as documentation describing what is (not) allowed by APIs and hence facilitate code reuse and programmers' communication. However, a JSON schema as an API documentation is quite different from documentation considered in this dissertation so far in that it uses formal types, i.e., it uses a set of predefined data types and each type has its own set of validation keywords which can restrict the values a type may inhibit.

Because schema validation can happen only once concrete data occurs during an execution, *data compatibility bugs* may be detected too late or not at all. Examples include evolving the schema for a web API, which may unexpectedly break client applications, or accidentally running a machine learning pipeline on incorrect data. This chapter presents a novel way of detecting a class of data compatibility bugs by reasoning about JSON schemas, a form of API documentation, via *subschema checking*. Subschema checks can find bugs before concrete JSON data is available and across all possible data specified by a schema. This chapter presents a formal algorithm, similar to that of a subtype checker and not a learning-based approach, to leverage the formal nature of JSON Schema.

5.1 MOTIVATION

JSON is a data serialization format that is widely adopted to store data on disk or send it over the network. The format supports primitive data types, such as strings, numbers, and Booleans, and two possibly nested data structures: arrays, which represent sorted lists of values, and objects, which represent unsorted maps of key-value pairs. JSON is used in numerous applications. It is the most popular data exchange format in web APIs, ahead of XML [Rod+16]. Cloud-hosted applications also use JSON pervasively, e.g., in micro-services that communicate via JSON data [New15]. On the data storage side, not only do traditional database management systems, such as Oracle, IBM DB2, MySQL, and PostgreSQL, now support JSON, but two of the most widely deployed NoSQL database management systems, MongoDB and CouchDB/Cloudant, are entirely based on JSON [RW12]. Beyond these applications, JSON is also gaining adoption in machine learning [Hir+19; Smi+19].

With the broad adoption of JSON as a data serialization format soon emerged the need for a way to describe how JSON data should look. For example, a web API that consumes JSON data can avoid unexpected behavior if it knows the structure of the data it receives. *JSON Schema* declaratively defines the structure of nested values (JSON data) via types (JSON schemas) [Pez+16]. A JSON Schema *validator* checks whether JSON data d conforms to a schema s. JSON Schema validators exist for many programming languages and are widely used to make software more reliable [Zyp09].

Despite the availability of JSON schema validators, some data-related bugs may get exposed late in the development process or even remain unnoticed until runtime misbehavior is observed. As one example, consider a RESTful web API for which the data types are specified with JSON schemas. If the API, and hence the schemas, evolve, the revised schemas may not be backward compatible and unexpectedly break client applications [EZG15; Li+13]. As another example, consider a machine learning pipeline where the data, as well as the input and output of operations, are specified with JSON schemas [Hir+19; Smi+19]. If some data is incorrect or different components of the pipeline have incompatible expectations, the pipeline may compute incorrect results or crash after hours of computation. For both of the above scenarios, schema validators can detect these problems only at runtime, because that is when concrete JSON data is available for validation. Even worse, the problems may remain completely unnoticed until some data that

triggers the problem occurs. We call such problems *data compatibility bugs*, which means that two pieces of software that share some JSON data have incompatible expectations about the data.

This chapter presents a novel way of detecting this class of data compatibility bugs early. The key idea is to check for two given JSON schemas whether one schema is a subschema (subtype) of the other schema. Because such a check can be performed on the schema-level, i.e., independently of concrete JSON data, the approach can detect data compatibility bugs earlier and more reliably than JSON schema validation alone. For the first of the above examples, our approach can check whether a revised schema is a subtype or a supertype of the original schema. If it is neither of the two, then the schema evolution is a breaking API change that should be communicated accordingly to clients. For the second example, the approach can check if the schema of the input given to an ML pipeline step is a subtype of the schema expected by that step, which reveals bugs before running the pipeline.

Deciding whether one JSON schema is a subtype of another is far from trivial because JSON Schema is a surprisingly complex language. Semantically equivalent schemas can look syntactically dissimilar. Schemas for primitive types involve sophisticated features, such as regular expressions for strings, that interact with other features, such as string length constraints. JSON Schema supports enumerations (singleton types) and logic connectives (conjunction, disjunction, and negation) between schemas of heterogeneous types. Even schemas without explicit logic connectives often have implicit conjunctions and disjunctions. JSON schemas or their nested fragments thereof can be uninhabited. As a result of these and other features of JSON Schema, simple structural comparison of schemas is insufficient to address the subschema question.

Our work addresses these challenges based on the insight that the subschema problem can be decomposed into simpler subproblems. Given two JSON schemas, the approach first canonicalizes and then simplifies the schemas using a series of transformations. While preserving the semantics of the schemas, these transformations reduce the number of cases and ensure that the schemas consist of schema fragments that each describe a single basic type of JSON data. This homogeneity enables the last step, recursively performing the subtype check using type-specific checkers. We have built an open-source tool, jsonSubSchema, that always terminates, returning one of three answers for a subschema check: true, false, or un-

known. It returns unknown for a small set of rarely occurring features of schemas. When it returns true or false, it is always correct.

We evaluate the approach with 11,478 pairs of real-world JSON schemas gathered from different domains, including schemas used for specifying web APIs, cloud computing, and machine learning pipelines. The results show that the approach decides the subschema question for 96% of all schemas, clearly outperforming the closest existing tool in terms of both precision and recall. Applying JSON subschema checking for bug detection, we find 43 data compatibility bugs related to API evolution and mistakes in machine learning pipelines. All bugs are confirmed by developers and 38 are already fixed. Developers acknowledged our work and emphasized the severity of many of the bugs.

5.2 PROBLEM STATEMENT

In this section we give a brief background on JSON Schema and then introduce the subschema problem and the several challenges it faces.

5.2.1 *Background*

JSON Schema is a declarative language for defining the structure and permitted values of JSON data [Zyp09]. This work focuses on JSON Schema draft-04 [GZ13], one of the most widely adopted versions.

JSON Schema itself uses JSON syntax. To specify which data types are allowed, JSON Schema uses the keyword `type` with one type name or a list of type names. For example, the schema `{'type':'string'}` accepts strings, whereas the schema `{'type':['null', 'boolean']}` accepts null or Boolean values. Each JSON type has a set of validation *keywords* that restrict the values a schema of this type permits. For example, `{'type':'integer', 'minimum':0}` restricts integers to be non-negative, whereas `{'type':'string', 'pattern':'^[A-Za-z0-9]+$'}` uses the keyword `pattern` with a regular expression to restricts strings to be alphanumeric.

In addition to type-specific keywords, JSON Schema allows enumerating exact values with `enum` and combining different schemas using a set of logic connectives. For example, schema `{'enum':['a', [], 1]}` restricts the set of permitted JSON values to the string literal `'a'`, an empty array, or the integer 1. Logic connectives, such as `anyOf`, `allOf`, and `not`, allow schema writers to express disjunctions, conjunctions, and negations of schemas. The

empty schema, `{}`, is the top of the schema hierarchy, i.e., any JSON data is valid for `{}`. The negation of the empty schema, `{'not':{}}`, is the bottom of the hierarchy, i.e., no JSON data is valid for it. Finally, the keyword `$ref` retrieves schemas using URIs and JSON pointers. JSON validation against a schema with `$ref` has to satisfy the schema retrieved from the specified URI or JSON pointer.

Figure 5.1 shows the full grammar of JSON schemas. The start symbol of the grammar is *schema*. A schema can mix keywords for all types as well as logic connectives. All-caps indicates literal tokens such as *NUM* or *BOOL* whose lexical syntax follows the usual conventions of JSON. We refer the interested reader to the full specification of JSON Schema [GZ13] and its formalization [Pez+16].

5.2.2 *JSON Subschema Problem*

This chapter presents how to detect data compatibility bugs by addressing the JSON subschema problem. Suppose a schema validator that determines whether JSON data d is valid according to a schema s: $valid(d, s) \rightarrow \{True, False\}$.

Definition 5.2.1 (JSON Subschema) *For any two JSON schemas s and t. Schema s is a subschema (subtype) of schema t, denoted $s <: t$, if and only if:*

$$\forall d : valid(d, s) \implies valid(d, t)$$

The subschema relation is a form of subtyping that views a type (JSON schema) as a set of values (JSON data) [PSW76].

As an example, consider the schema shown in Figure 5.2a, an excerpt of version 0.6.1 of a real-world schema that describes an API.[1] The schema evolves into version 0.6.2 (Figure 5.2b). Both schemas describe an object with a property `'category'` with a fixed set of values. Version 0.6.1 is a subschema of version 0.6.2 because all documents valid according to the first schema are also valid according to the second schema. In contrast, version 0.6.2 is not a subschema of version 0.6.1 because the JSON datum `{'category':'stock'}` is valid with respect to version 0.6.2 but not version 0.6.1. To retain backward compatibility, this evolution is fine for API arguments, but it could break clients if the schema describes an API response.

1 From a collection of schemas for content used by the Washington Post [Wp2]

```
schema     ::= {type?, strKw, numKw, arrKw, objKw,
                enum?, not?, allOf?, anyOf?, oneOf?, ref?}
type       ::= "type": (typeName | [ typeName+ ])
typeName   ::= "null" | "boolean" | "string" | "number" | "integer" |
               "array" | "object"
strKw      ::= minLength?, maxLength?, pattern?
minLength  ::= "minLength": NUM
maxLength  ::= "maxLength": NUM
pattern    ::= "pattern": REGEX
numKw      ::= minimum?, maximum?, exclMin?, exclMax?, multOf?
minimum    ::= "minimum": NUM
maximum    ::= "maximum": NUM
exclMin    ::= "exclusiveMinimum": BOOL
exclMax    ::= "exclusiveMaximum": BOOL
multOf     ::= "multipleOf": NUM
arrKw      ::= items?, minItems?, maxItems?, addItems?, uniqItems?, contains?
items      ::= "items": (schema | [ schema+ ])
minItems   ::= "minItems": NUM
maxItems   ::= "maxItems": NUM
addItems   ::= "additionalItems": (BOOL | schema)
uniqItems  ::= "uniqueItems": BOOL
contains   ::= "contains": schema
objKw      ::= props?, minProps?, maxProps?, required?, addProps?,
               patProps?, depend?
props      ::= "properties": {(STR :schema)* }
minProps   ::= "minProperties": NUM
maxProps   ::= "maxProperties": NUM
required   ::= "required": [STR* ]
addProps   ::= "additionalProperties": (BOOL|schema)
patProps   ::= "patternProperties": {(REGEX :schema)* }
depend     ::= "dependencies": {(STR :(schema | [ STR+ ]))* }
enum       ::= "enum": [ VALUE+ ]
not        ::= "not": schema
allOf      ::= "allOf": [ schema+ ]
anyOf      ::= "anyOf": [ schema+ ]
oneOf      ::= "oneOf": [ schema+ ]
ref        ::= "$ref": PATH
```

FIGURE 5.1: Grammar of full JSON Schema (draft-04).

```
{'type': 'object',
 'properties': {
   'category': {
      'type': 'string',
      'enum': ['staff', 'wires',
               'other']}}}
```

(a) Version 0.6.1

```
{'type': 'object',
 'properties': {
   'category': {
      'type': 'string',
      'enum': ['staff', 'wires',
               'stock', 'other']}}}
```

(b) Version 0.6.2

FIGURE 5.2: A real example of schema evolution from the Washington Post content management system.

5.2.3 *Challenges*

The rich feature set of JSON Schema makes establishing or refuting a subtype relation between two schemas non-trivial. Even for simple, structurally similar schemas, such as `{'enum':[1, 2]}` and `{'enum':[2, 1]}`, equivalence does not hold through textual equality. There are several challenges for algorithmically checking the JSON schema subtype relation.

First, the schema language is flexible and the same set of JSON values, i.e., the same type, could be described in several different syntactical forms, i.e., schemas. For example, Figure 5.3 shows five equivalent schemas describing a JSON datum that is either a non-empty string or null.

Second, even for primitive types, such as strings and numbers, nominal subtyping is not applicable. JSON Schema lets users specify various constraints on primitive types, resulting in non-trivial interactions that are not captured by nominal types. For example, one cannot infer that an `integer` schema is a subtype of a `number` schema without properly comparing the range and multiplicity constraints of the schemas.

Third, logic connectives combine non-homogeneous types, e.g., `string` and `null` in Figure 5.3b. Moreover, enumerations restrict types to predefined values, which require careful handling, especially when enumerations interact with non-enumerative types, such as in Figures 5.3a, 5.3b, and 5.3c.

```
{'type': ['null', 'string'],
 'not': {'enum': ['']}}
```

(a)

```
{'anyOf': [
   {'type': 'null'},
   {'type': 'string'}],
 'not': {'type': 'string', 'enum': ['']}}
```

(b)

```
{'allOf': [
   {'anyOf': [
      {'type': 'null'},
      {'type': 'string'}]},
   {'not': {'type': 'string', 'enum': ['']}}]}
```

(c)

```
{'anyOf': [
   {'type': 'null'},
   {'type': 'string', 'pattern': '.+'}]}
```

(d)

```
{'allOf': [
   {'anyOf': [
      {'type': 'null'},
      {'type': 'string'}]},
   {'anyOf': [
      {'type': 'boolean'}, {'type': 'null'},
      {'type': 'number'}, {'type': 'integer'},
      {'type': 'array'}, {'type': 'object'},
      {'type': 'string', 'pattern': '.+'}]}]}
```

(e)

FIGURE 5.3: Five syntactically different but semantically equivalent JSON schemas describing a JSON value that is either a non-empty string or null.

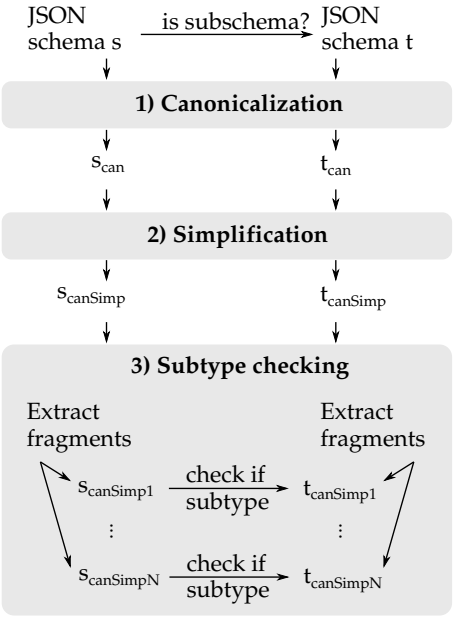


FIGURE 5.4: Overview of JSON subschema checker.

Fourth, the schema language allows implicit conjunctions and disjunctions. For example, Figure 5.3b has an implicit top-level conjunction between the subschemas under `anyOf` and `not`. As another example, a schema that lacks a `type` keyword, such as `{'pattern':'.+'}`, has an implicit disjunction of all possible types, while still enforcing any type-specific keyword, such as the pattern for strings only. Figure 5.3e makes this implicit disjunction explicit.

Finally, JSON Schema allows uninhabited types. That is, a schema can be syntactically valid yet semantically self-contradicting, i.e., it does not validate any data, e.g., `{'type':'number', 'minimum':5, 'maximum':0}` and `{'type':'boolean', 'enum':[1, 2, 3]}`. Such schemas validate no JSON value at all and complicate reasoning about subtyping.

5.3 ALGORITHM

This section describes how we address the problem of checking whether one JSON schema is a subtype of another. Because JSON schemas are complex, creating a subtype checker for arbitrary schemas directly would necessitate

a complex algorithm to handle all of its variability. A key insight of our work is to instead decompose the problem into three steps, outlined in Figure 5.4. The first step canonicalizes a given schema into an equivalent but more standardized schema (Section 5.3.1). The second step further simplifies a schema by eliminating enumerations, negation, intersection, and union of schemas where possible (Section 5.3.2). Table 5.1 summarizes the first two steps. Finally, the third step checks for two canonicalized and simplified schemas whether one is a subtype of the other by extracting and comparing type-homogeneous schema fragments (Section 5.3.3).

NOTATION We formalize canonicalization and simplification via rewrite rules of the form $s_1 \rightarrow s_2$. The notation $s.k$ indicates access of property k in schema s. For any JSON schema s, helper function $dom(s)$ returns its property names, i.e., the set of keys in the key-value map s. The notation $s[k \mapsto v]$ indicates a substitution, which is a copy of s except that the mapping of key k is added or changed to value v. The notation $[\ldots]$ indicates a JSON array and the notation $\{\ldots\}$ indicates a JSON object. The notation $\{k\!:\!v \mid \ldots\}$ indicates a JSON object comprehension. The notation $a \parallel b$ is a default operator that returns a if it is defined and b otherwise.

5.3.1 *JSON Schema Canonicalization*

This section introduces a canonicalization procedure that compiles any JSON schema into an equivalent canonical schema. The canonicalization enforces two main simplifications. First, JSON Schema allows schemas to mix specifications of different types. To enable local, domain-specific reasoning in the subtype checker, canonicalization first splits up these schemas into smaller, homogeneously typed schemas combined with logic connectives. Second, JSON Schema also allows many alternative ways to represent the same thing. Additionally, most keywords can be omitted and defaults assumed. Canonicalization picks, when possible, one form, and explicates omitted defaults. Column "Canonicalized" of Table 5.1 summarizes the properties that the canonicalizer establishes. Given any JSON schema as input, canonicalization terminates and produces a semantically equivalent, canonical JSON schema as output.

TYPE-INDEPENDENT TRANSFORMATIONS Figure 5.5 presents non-type-specific canonicalization rules whose purpose is to enable reasoning about one type or connective at a time. Rule *multiple types* applies to schemas

TABLE 5.1: Properties of original, canonicalized (Section 5.3.1), and simplified (Section 5.3.2) schemas.

Language feature	Use of feature in schemas		
	Full JSON Schema	Canonicalized	Simplified
`null`	(no keywords)	Yes	Yes
`boolean`	(no keywords)	Represented as `enum`	Represented as `enum`
`string`	`{min,max}Length`, `pattern`	Keyword `pattern` only	Keyword `pattern` only
`number`	`{min,max}imum`, `multipleOf`, `exclusive{Min,Max}imum`	All keywords	All keywords
`integer`	(same keywords as number)	Eliminated	Eliminated
`array`	`{min,max}Items`, `items`, `additionalItems`, `uniqueItems`	All keywords, but `items` is always a list and `additionalItems` is always a schema	All keywords, but `items` is always a list and `additionalItems` is always a schema
`object`	`properties`, `{min,max}Properties`, `required`, `patternProperties`, `additionalProperties`, `dependencies`	Only `{min,max}Properties`, `patternProperties`, `required` keywords	Only `{min,max}Properties`, `patternProperties`, `required` keywords
`enum`	Heterogeneous, any type	Homogeneous, any type	Only for `boolean`
`not`	Multiple connectives	Single connective	Only for `number`, `array`, `object`
`allOf`	Multiple connectives	Single connective	Only for `not`
`anyOf`	Multiple connectives	Single connective	Only for `not`, `allOf`, `array`, `object`, and disjoint `number`
`oneOf`	Multiple connectives	Single connective	Eliminated

$$\text{multiple types}\ \frac{s.\texttt{type} = [\tau_1, \ldots, \tau_n]}{s \to \{\texttt{anyOf}\colon [s[\texttt{type} \mapsto \tau_1], \ldots, s[\texttt{type} \mapsto \tau_n]]\}}$$

$$\text{multiple connectives}\ \frac{\begin{array}{c} dom(s) \cap \{\texttt{enum}, \texttt{anyOf}, \texttt{allOf}, \texttt{oneOf}, \texttt{not}\} \neq \emptyset \\ dom(s) \setminus \{c\} \neq \emptyset \end{array}}{s \to \{\texttt{allOf}\colon [\{c\colon s.c\}, \{k\colon s.k \mid k \in (dom(s) \setminus \{c\})\}]\}}$$

$$\text{missing type}\ \frac{dom(s) \cap \{\texttt{type}, \texttt{enum}, \texttt{anyOf}, \texttt{allOf}, \texttt{oneOf}, \texttt{not}\} = \emptyset}{s \to s[\texttt{type} \mapsto \mathit{Jtypes}]}$$

FIGURE 5.5: Non-type-specific canonicalization rules that ensure exactly one type, enum, or logic connective.

whose type is a list, such as in the example in Figure 5.3a, making the implicit disjunction explicit using `anyOf`, as shown in Figure 5.3b. Rule *multiple connectives* applies to schemas that contain a connective mixed with other connectives, such as in the example in Figure 5.3b, making the implicit conjunction explicit using `allOf`, as shown in Figure 5.3c. Rule *missing type* generously assumes all JSON types are possible, yielding an implicit disjunction to be further canonicalized by the multiple-types rule.

TYPE-DEPENDENT TRANSFORMATIONS Figure 5.6 presents type-specific canonicalization rules whose purpose is to reduce the number of cases to handle for later simplification and subschema rules. Rule *missing keyword* adds the default for a keyword if there is a single type and the keyword for that type is missing, using a helper function *default* that returns the default from the meta-schema in Figure 5.1 and maps numerical and length constraints of different types to appropriate representations, e.g., `minimum` to $-\infty$, `maximum` to ∞, and `minLength` to 0, for convenience. Rule *irrelevant keywords* strips out spurious keywords that do not apply to a given type (or to any type), using a helper function *kw* that returns the relevant keywords in Figure 5.1.

ELIMINATING INTEGERS AND ONEOF Rule *integer* rewrites integer schemas to number schemas with the appropriate `multipleOf`. For instance, the schema `{'type':'integer}` is rewritten into `{'type':'number', 'multipleOf':1}`. Rule *oneOf* eliminates the `oneOf` keyword by rewriting the exclusive or into a disjunction of conjunctions.

CANONICALIZING ENUMERATIONS In full JSON Schema, enumerations of values may be heterogeneous, i.e., contain multiple different types. Our canonicalization ensures that enumerations are homogeneous, so that each enumeration schema contains values of a single type. To this end, rule *heterogeneous enum* transforms any heterogeneous enumeration into a disjunction of multiple homogeneous enumerations using a helper function *typeOf* that maps a concrete JSON value to its type.

REPRESENTING STRINGS AS PATTERNS In full JSON Schema, strings may be restricted based on their minimum length, maximum length, and a regular expression. To reduce the number of cases, and since length constraints interact with the pattern, if specified, `minLength` and `maxLength` keywords are transformed into a semantically equivalent regular expression (intersected with the `pattern` schema), so canonicalized string schemas only have the keyword `pattern`. The two *string* rules in Figure 5.6 show these transformations.

CANONICALIZING ARRAYS We canonicalize schemas that describe arrays using two transformations that reduce the number of ways in which the `items` and `additionalItems` keywords may be used. The two *array* rules handle keywords that can be specified in multiple different ways, as indicated by meta-schemas with `anyOf` in Figure 5.1. Rule *array with one schema for all items* changes the keyword `items` from a single schema to a list (empty) of schemas, by moving the schema into `additionalItems`. For example, the transformation would perform the following:

```
{'type': 'array',                    →   {'type': 'array',
 'items': {'type': 'number'}}             'additionalItems': {'type': 'number'}}
```

Since `additionalItems` may be either a schema or a Boolean (`false` disallows additional items), the second transformation replaces a Boolean `additionalItems` with a corresponding JSON schema, where the schemas `{}` and `{'not':{}}` replace `true` and `false`, respectively. So `additionalItems` becomes always a schema. Rule *array with additionalItems false* changes the keyword `additionalItems` from a boolean to a schema (bottom).

CANONICALIZING OBJECTS Schemas for objects have various keywords. The five *object* rules eliminate the keywords `dependencies`, `properties`, and `additionalProperties` by rewriting them into the keywords `required` and `patternProperties`, and ensure that the patterns in `patternProperties` use non-overlapping regular expressions. The object rules are the most intricate

out of the canonicalization rules because in JSON Schema, object schemas have the largest number of special cases. Reducing the cases reduces the complexity of subsequent rules for simplification and subschema checking.

Rule *object with additionalProperties false* changes the `additionalProperties` keyword from a boolean to a schema (bottom). Rule *object with properties* eliminates `properties` and `additionalProperties` by rewriting them into `patternProperties`. First, it turns each property key k_i from `properties` into a pattern property with the regular expression `'^`k_i`$'` that accepts exactly k_i. Second, it subtracts all keys k_i from each of the original pattern properties so they only apply as a fall-back, where the notation $p_1 \setminus p_2$ indicates regular expression subtraction. Third, it creates a tertiary fall-back regular expression that applies when neither the original properties nor the original pattern properties match, using the notation $\neg p$ for the complement of a regular expression, and associates that regular expression with the original `additionalProperties`. Finally, it removes the eliminated keywords `properties` and `additionalProperties` from the resulting schema.

Rule *object with string list dependencies*, when iterated, eliminates dependencies specified as a list of property names by rewriting them into dependencies specified as a schema. Rule *object with schema dependencies*, when iterated, eliminates dependencies of a key k_i on a schema s_i by rewriting them into a conjunction with either s_i or a schema that enforces the absence of k_i. Rule *object with overlapping pattern properties* rewrites a pair of pattern properties with overlapping regular expressions p_i and p_j so their regular expressions match only disjoint keys, by replacing them with different schemas for the cases where (i) both p_i and p_j match, (ii) only p_i and not p_j matches, and (iii) only p_j and not p_i matches. When iterated, this eliminates all overlapping patterns, facilitating local reasoning. For example, this object schema would be canonicalized as follows:

```
{'type':'object',                          {'type':'object',
 'properties': {                            'patternProperties': {
   'a': {'type':'string'},          →         '^a$': {'type':'string'},
   'b': {'type':'array'}}                    '^b$': {'type':'array'},
 'patternProperties': {                      '([^a]+a!a.).*': {'type':'boolean'}}}
   'a': {'type':'boolean'}}}
```

The non-canonical schema on the left describes the types of properties "a" and "b" using `properties`, and of any property that contains an "a" using `additionalProperties`. The canonical schema on the right describes the same type constraints by expressing all property names as regular expressions, which will simplify the subschema check.

$$\text{missing keyword}\ \frac{s.\texttt{type} = \tau \quad \tau \in \mathit{Jtypes} \quad \tau \neq \texttt{string} \quad k \in \mathit{kw}(\tau) \quad k \notin \mathit{dom}(s)}{s \to s[k \mapsto \mathit{default}(k)]}$$

$$\text{irrelevant keywords}\ \frac{s.\texttt{type} = \tau \quad \tau \in \mathit{Jtypes}}{s \to \{k\texttt{:}s.k \mid k \in (\mathit{dom}(s) \cap (\mathit{kw}(\tau) \cup \{\texttt{type}, \texttt{enum}\})\}}$$

$$\text{integer}\ \frac{s.\texttt{type} = \texttt{integer}}{s \to s[\texttt{type} \mapsto \texttt{number}, \texttt{multipleOf} \mapsto \mathit{lcm}(1, s.\texttt{multipleOf} \parallel 1)]}$$

$$\text{oneOf}\ \frac{s.\texttt{oneOf} = [s_1, \ldots, s_n]}{\begin{array}{l} s \to \{\texttt{anyOf}\texttt{:}[\{\texttt{allOf}\texttt{:}[s_1, \{\texttt{not}\texttt{:}s_2\}, \ldots, \{\texttt{not}\texttt{:}s_n\}]\}, \\ \qquad \ldots, \\ \qquad \{\texttt{allOf}\texttt{:}[\{\texttt{not}\texttt{:}s_1\}, \ldots, \{\texttt{not}\texttt{:}s_{n-1}\}, s_n]\}]\} \end{array}}$$

$$\text{heterogeneous enum}\ \frac{s.\texttt{enum} = [v_1, \ldots, v_n] \quad \exists j, \mathit{typeOf}(v_j) \neq \mathit{typeOf}(v_1)}{\begin{array}{l} s \to \{\texttt{anyOf}\texttt{:}[s[\texttt{enum} \mapsto [v_i \mid \mathit{typeOf}(v_i) = \mathit{typeOf}(v_1)], \texttt{type} \mapsto \mathit{typeOf}(v_1)], \\ \qquad s[\texttt{enum} \mapsto [v_j \mid \mathit{typeOf}(v_j) \neq \mathit{typeOf}(v_1)]]\} \end{array}}$$

FIGURE 5.6: Type-specific canonicalization rules.

(Continued on next page)

$$\text{string without maxlength}\quad \frac{s.\texttt{type} = \texttt{string} \quad s.\texttt{pattern} = p \quad s.\texttt{minLength} = a \quad \texttt{maxLength} \notin dom(s)}{s \to \{\texttt{type:string}, \texttt{pattern:}p \cap \texttt{'\^{}.\{}a\texttt{\}'}\}}$$

$$\text{string with maxlength}\quad \frac{s.\texttt{type} = \texttt{string} \quad s.\texttt{pattern} = p \quad s.\texttt{minLength} = a \quad s.\texttt{maxLength} = b}{s \to \{\texttt{type:string}, \texttt{pattern:}p \cap \texttt{'\^{}.\{}a,b\texttt{\}\$'}\}}$$

$$\text{array with one schema for all items}\quad \frac{s.\texttt{type} = \texttt{array} \quad s.\texttt{items} = \{\ldots\}}{s \to s[\texttt{items} \mapsto [], \texttt{additionalItems} \mapsto s.\texttt{items}]}$$

$$\text{array with additionalItems false}\quad \frac{s.\texttt{type} = \texttt{array} \quad s.\texttt{additionalItems} = \texttt{false}}{s \to s[\texttt{additionalItems} \mapsto \{\texttt{not:}\{\}\}]}$$

FIGURE 5.6: Type-specific canonicalization rules.

(Continued on next page)

$$\text{object with additionalProperties false}\quad \frac{s.\texttt{type} = \texttt{object} \qquad s.\texttt{additionalProperties} = \texttt{false}}{s \to s[\texttt{additionalProperties} \mapsto \{\texttt{not}:\{\}\}]}$$

$$\text{object with properties}\quad \frac{\begin{array}{c} s.\texttt{type} = \texttt{object} \qquad\qquad s.\texttt{properties} = \{k_1 : s_{k_1}, \ldots, k_n : s_{k_n}\} \\ s.\texttt{additionalProperties} = \{\ldots\} \qquad s.\texttt{patternProperties} = \{p_1 : s_{p_1}, \ldots, p_m : s_{p_m}\} \end{array}}{\begin{array}{l} s \to s[\texttt{patternProperties} \mapsto \{\texttt{'\^{}}k_1\texttt{\$'}:s_{k_1}, \quad \ldots, \quad \texttt{'\^{}}k_n\texttt{\$'}:s_{k_n}, \\ \qquad\qquad p_1 \setminus \texttt{'\^{}(}k_1\texttt{|}\ldots\texttt{|}k_n\texttt{)\$'}:s_{p_1}, \quad \ldots, \quad p_m \setminus \texttt{'\^{}(}k_1\texttt{|}\ldots\texttt{|}k_n\texttt{)\$'}:s_{p_m}, \\ \qquad\qquad \neg\texttt{'\^{}(}k_1\texttt{|}\ldots\texttt{|}k_n\texttt{)\$|}p_1\texttt{|}\ldots\texttt{|}p_m\texttt{'}:s.\texttt{additionalProperties}\}] \\ \quad \setminus \{\texttt{properties}, \texttt{additionalProperties}\} \end{array}}$$

$$\text{object with string list dependencies}\quad \frac{s.\texttt{type} = \texttt{object} \qquad s.\texttt{dependencies} = \{k_i : [k_{i_1}, \ldots, k_{i_n}]\} \cup d_{rest}}{s \to s[\texttt{dependencies} \mapsto d_{rest} \cup \{k_i : \{\texttt{type}:\texttt{object}, \texttt{required}:[k_{i_1}, \ldots, k_{i_n}]\}\}]}$$

$$\text{object with schema dependencies}\quad \frac{s.\texttt{type} = \texttt{object} \qquad s.\texttt{dependencies} = \{k_i : s_i\} \cup d_{rest}}{\begin{array}{l} s \to \{\texttt{allOf}:[s[\texttt{dependencies} \mapsto d_{rest}], \\ \qquad\qquad \{\texttt{anyOf}:[s_i, \{\texttt{type}:\texttt{object}, \texttt{properties}:\{k_i:\texttt{not}:\{\}\}\}]\}\} \end{array}}$$

$$\text{object with overlapping patternProperties}\quad \frac{s.\texttt{type} = \texttt{object} \qquad s.\texttt{patternProperties} = \{p_i : s_i\} \cup \{p_j : s_j\} \cup d_{rest}}{\begin{array}{l} s \to s[\texttt{patternProperties} \mapsto \\ \qquad \{p_i \cap p_j : \{\texttt{allOf}:[s_i, s_j]\}\} \cup \{p_i \cap \neg p_j : s_i\} \cup \{\neg p_i \cap p_j : s_j\} \cup d_{rest}] \end{array}}$$

FIGURE 5.6: Type-specific canonicalization rules.

5.3.2 *Simplification of Canonicalized Schemas*

This section describes the second step of our approach: a simplifier that compiles any canonical JSON schema into an equivalent simplified schema. Column "Simplified" of Table 5.1 summarizes the properties that the simplifier establishes. The simplifier eliminates many cases of `enum`, `not`, `allOf`, and `anyOf` connectives, thus making subschema checking rules less complicated. Unfortunately, in some cases, JSON schema cannot express schemas without these connectives, so they cannot be completely simplified away.

SIMPLIFICATION OF ENUMERATIONS Figure 5.8 shows simplification rules for `enum`, which turn schemas with enums into schemas without enums by using restrictions keywords from their corresponding types instead. Rule *multi-valued enum* puts each non-Boolean enumerated value into an `enum` of its own. The rules for primitive types (`null`, `string`, and `number`) express a primitive enumerated value via a schema that does not use an `enum`. For instance, in Figure 5.3c, the enumerated empty string value is compiled into the regular expression `'^$'` before computing its complement `'.+'` in Figure 5.3e. The rules for structured types (`array` and `object`) push down `enums` to components; with iteration, the rules eventually reach primitive types and the `enums` get eliminated.

The simplifier does not eliminate Boolean enumerations as the space of values is finite and there is no other way to specify the true or false value.

SIMPLIFICATION OF NEGATED SCHEMAS Figure 5.7 shows simplification rules for `not`, which eliminates negation except for numbers, arrays, and objects. Rule *not type* turns a schema of a given type τ into a disjunction of either $\neg s$ (the complement of the values permitted by s in τ) or values of any type other than τ. An example for this rule in action is the rewrite from Figure 5.3c to Figure 5.3e, where the complement of a string schema introduces schemas of all non-string types. The *complement* rules for `null`, `boolean`, and `string` use the bottom type `{'not':{}}`, the complement of the Boolean enumeration, and the complement of the regular expression, respectively. Rules *not anyOf* and *not allOf* use De Morgan's theorem to push negation through disjunction and conjunction, and rule *not not* eliminates double negation. Unfortunately, JSON Schema is not closed under complement for numbers, arrays, and objects. For example, the complement of schema `{'type':'number', 'multipleOf':1}` is $\mathbb{R} \setminus \mathbb{Z}$, which cannot be expressed in JSON schema without a negation. Similar counter-examples

$$\text{not type}\ \frac{s.\texttt{type} = \tau \qquad \tau \in \mathit{Jtypes}}{\{\texttt{not}\!:\!s\} \to \{\texttt{anyOf}\!:\![\neg s, \{\texttt{type}\!:\!(\mathit{Jtypes} \setminus \tau)\}]\}}$$

$$\begin{array}{l}\text{complement}\\ \text{boolean}\end{array}\ \frac{s.\texttt{type} = \texttt{boolean} \qquad s.\texttt{enum} = e}{\neg s \to \{\texttt{type}\!:\!\texttt{boolean}, \texttt{enum}\!:\!\neg e\}}$$

$$\begin{array}{l}\text{complement}\\ \text{string}\end{array}\ \frac{s.\texttt{type} = \texttt{string} \qquad s.\texttt{pattern} = p}{\neg s \to \{\texttt{type}\!:\!\texttt{string}, \texttt{pattern}\!:\!\neg p\}}$$

$$\text{not anyOf}\ \frac{s = \{\texttt{anyOf}\!:\![s_1, \ldots, s_n]\}}{\{\texttt{not}\!:\!s\} \to \{\texttt{allOf}\!:\![\{\texttt{not}\!:\!s_1\}, \ldots, \{\texttt{not}\!:\!s_n\}]\}}$$

$$\text{not allOf}\ \frac{s = \{\texttt{allOf}\!:\![s_1, \ldots, s_n]\}}{\{\texttt{not}\!:\!s\} \to \{\texttt{anyOf}\!:\![\{\texttt{not}\!:\!s_1\}, \ldots, \{\texttt{not}\!:\!s_n\}]\}}$$

$$\text{not not}\ \frac{s = \{\texttt{not}\!:\!s_1\}}{\{\texttt{not}\!:\!s\} \to s_1}$$

FIGURE 5.7: Simplification rules to eliminate negation, except for types `number`, `array`, and `object`.

exist for array and object schemas. The case of negated number schemas is handled later during subschema checking.

SIMPLIFICATION OF INTERSECTION OF SCHEMAS Figure 5.9 shows simplification rules for `allOf`. For example, the intersection type in Figure 5.3e yields the simplified schema in Figure 5.3d. Rule *singleton allOf* rewrites a conjunct of just one schema into that schema. Rule *fold allOf* turns an n-ary `allOf` into a binary one, so the remaining rules need to handle only the binary case. Rule *intersect heterogeneous types* returns the bottom type because intersection of incompatible types is the empty set, so the remaining rules only need to handle homogeneously-typed schemas. Rule *intersect null* rewrites two nulls to one null. Rule *intersect boolean* uses the intersection of enumerations. Rule *intersect string* uses the intersection of regular expressions. Rule *intersect number* uses helper functions *schema2range* and *range2schema* to convert back and forth between number schemas and mathematical ranges, and *lcm* to compute the least common multiple of the `multipleOf` constraints, where *lcm* handles undefined arguments by returning the other argument if defined, or an undefined value if both arguments are undefined. Rule *intersect array* takes advantage of the canonical form, where `items` are always specified as lists, to compute an item-wise

intersection; undefined per-item schemas default to `additionalItems`. Rule *intersect object* simply picks up the union of the `patternProperties` keywords, relying on the rule for objects with overlapping `patternProperties` to make them disjoint again later. Finally, rule *intersect anyOf* pushes conjunctions through disjunctions by using the distributivity of intersection over union. We choose not to push intersections through negations because we prefer the end result of simplification to resemble distributive normal form to the extent possible.

SIMPLIFICATION OF UNION OF SCHEMAS Figure 5.10 shows simplification rules for `anyOf`. In contrast to intersection, union allows incompatible types, e.g., `string` or `null` as in Figure 5.3d. Fortunately, such heterogeneous unions are non-overlapping for inhabited schemas, so they can be handled later during subschema checking. Rule *singleton anyOf* rewrites a disjunct of just one schema into that schema. Rule *fold anyOf* turns an n-ary `anyOf` into a binary one so the remaining rules only need to handle the binary case. Rule *union null* rewrites two nulls to one. Rule *union boolean* uses the union of enumerations. Rule *union string* uses the union of regular expressions. For example, the following schema that specifies strings using regular expressions gets simplified by computing the union of the two regular expressions:

```
{'anyOf': [
   {'type':'string', 'pattern':'.+'},   →   {'type':'string', 'pattern':'.+'}
   {'type':'string', 'pattern':'a'}]}
```

Rule *union number* turns a binary union with overlap into a ternary non-overlapping union. In other words, while it does not eliminate the union of number schemas, it does simplify subschema checking by at least making the union disjoint so it can be checked element-wise. Unfortunately, JSON Schema is not closed under union for types number, array, and object. For example, the union of `{'type':'number', 'minimum':0}` and `{'type':'number', 'multipleOf':1}` is $\mathbb{R}^+ \cup \mathbb{Z}$, which cannot be expressed in JSON schema without `anyOf`. There are similar counter-examples for arrays and objects. The case of unioned number schemas is handled later in subschema checking. As mentioned earlier, we would like simplification to end in schemas that resemble a distributive normal form, so we choose not to push `anyOf` through `allOf` or `not`.

$$\text{multi-valued enum}\ \frac{s.\texttt{type} = \tau \quad \tau \neq \texttt{boolean} \quad s.\texttt{enum} = [v_1, \ldots, v_n] \quad n > 1}{s \to \{\texttt{anyOf}\texttt{:}[\{\texttt{type}\texttt{:}\tau, \texttt{enum}\texttt{:}[v_1]\}, \ldots, \{\texttt{type}\texttt{:}\tau, \texttt{enum}\texttt{:}[v_n]\}]\}}$$

$$\text{null enum}\ \frac{s.\texttt{type} = \texttt{null} \quad s.\texttt{enum} = [\texttt{null}]}{s \to \{\texttt{type}\texttt{:}\texttt{null}\}}$$

$$\text{string enum}\ \frac{s.\texttt{type} = \texttt{string} \quad s.\texttt{enum} = [v]}{s \to \{\texttt{type}\texttt{:}\texttt{string}, \texttt{pattern}\texttt{:}\texttt{'}\hat{}v\$\texttt{'}\}}$$

$$\text{number enum}\ \frac{s.\texttt{type} = \texttt{number} \quad s.\texttt{enum} = [v]}{s \to \{\texttt{type}\texttt{:}\texttt{number}, \texttt{minimum}\texttt{:}v, \texttt{maximum}\texttt{:}v\}}$$

$$\text{array enum}\ \frac{s.\texttt{type} = \texttt{array} \quad s.\texttt{enum} = [[v_1, \ldots, v_n]]}{s \to \{\texttt{type}\texttt{:}\texttt{array}, \texttt{minItems}\texttt{:}n, \texttt{maxItems}\texttt{:}n, \texttt{items}\texttt{:}[\{\texttt{enum}\texttt{:}[v_1]\}, \ldots, \{\texttt{enum}\texttt{:}[v_n]\}]}$$

$$\text{object enum}\ \frac{s.\texttt{type} = \texttt{object} \quad s.\texttt{enum} = [\{k_1\texttt{:}v_1, \ldots, k_n\texttt{:}v_n\}]}{\begin{array}{c} s \to \{\texttt{type}\texttt{:}\texttt{object}, \texttt{required}\texttt{:}[k_1, \ldots, k_n], \texttt{additionalProperties}\texttt{:}\texttt{false}, \\ \texttt{properties}\texttt{:}\{k_1\texttt{:}\{\texttt{enum}\texttt{:}[v_1]\}, \ldots, k_n\texttt{:}\{\texttt{enum}\texttt{:}[v_n]\}\}\} \end{array}}$$

FIGURE 5.8: Simplification rules to eliminate `enum` except for type `boolean`.

$$\text{singleton allOf}\ \frac{s.\texttt{allOf} = [s_1]}{s \to s_1}$$

$$\text{fold allOf}\ \frac{s.\texttt{allOf} = [s_1, s_2, \ldots, s_n] \qquad n \geq 3}{s \to \{\texttt{allOf}\texttt{:}[s_1, \{\texttt{allOf}\texttt{:}[s_2, \ldots, s_n]\}]\}}$$

$$\text{intersect heterogeneous types}\ \frac{s_1.\texttt{type} \neq s_2.\texttt{type}}{\{\texttt{allOf}\texttt{:}[s_1, s_2]\} \to \{\texttt{not}\texttt{:}\{\}\}}$$

$$\text{intersect null}\ \frac{s_1.\texttt{type} = \texttt{null} \qquad s_2.\texttt{type} = \texttt{null}}{\{\texttt{allOf}\texttt{:}[s_1, s_2]\} \to \{\texttt{type}\texttt{:}\texttt{null}\}}$$

$$\text{intersect boolean}\ \frac{s_1.\texttt{type} = \texttt{boolean} \qquad s_2.\texttt{type} = \texttt{boolean}}{\{\texttt{allOf}\texttt{:}[s_1, s_2]\} \to \{\texttt{type}\texttt{:}\texttt{boolean}, \texttt{enum}\texttt{:}s_1.\texttt{enum} \cap s_2.\texttt{enum}\}}$$

$$\text{intersect string}\ \frac{s_1.\texttt{type} = \texttt{string} \qquad s_2.\texttt{type} = \texttt{string}}{\{\texttt{allOf}\texttt{:}[s_1, s_2]\} \to \{\texttt{type}\texttt{:}\texttt{string}, \texttt{pattern}\texttt{:}s_1.\texttt{pattern} \cap s_2.\texttt{pattern}\}}$$

$$\text{intersect number}\ \frac{s_1.\texttt{type} = \texttt{number} \qquad s_2.\texttt{type} = \texttt{number} \qquad r_1 = \mathit{schema2range}(s_1) \qquad r_2 = \mathit{schema2range}(s_2)}{\{\texttt{allOf}\texttt{:}[s_1, s_2]\} \to \mathit{range2schema}(r_1 \cap r_2) \cup \{\texttt{multipleOf}\texttt{:}\mathit{lcm}(s_1.\texttt{multipleOf}, s_2.\texttt{multipleOf})\}}$$

FIGURE 5.9: Simplification rules to eliminate `allOf` except for connective `not`.

(Continued on next page)

intersect array

$$\frac{s_1.\texttt{type} = \texttt{array} \quad s_2.\texttt{type} = \texttt{array} \quad s_1.\texttt{items} = [s_{1_1}, \ldots, s_{1_k}] \quad s_2.\texttt{items} = [s_{2_1}, \ldots, s_{2_m}] \quad n = max(k, m)}{\begin{array}{l}\{\texttt{allOf}:[s_1, s_2]\} \rightarrow \{\texttt{type}:\texttt{array}, \\ \quad \texttt{minItems}:max(s_1.\texttt{minItems}, s_2.\texttt{minItems}), \\ \quad \texttt{maxItems}:min(s_1.\texttt{maxItems}, s_2.\texttt{maxItems}), \\ \quad \texttt{items}:[\{\texttt{allOf}:[s_{1_1} \parallel s_1.\texttt{additionalItems}, s_{2_1} \parallel s_2.\texttt{additionalItems}]\}, \\ \qquad \ldots, \\ \qquad \{\texttt{allOf}:[s_{1_n} \parallel s_1.\texttt{additionalItems}, s_{2_n} \parallel s_2.\texttt{additionalItems}]\}], \\ \quad \texttt{additionalItems}:\{\texttt{allOf}:[s_1.\texttt{additionalItems}, s_2.\texttt{additionalItems}]\}, \\ \quad \texttt{uniqueItems}:s_1.\texttt{uniqueItems} \wedge s_2.\texttt{uniqueItems}\}\end{array}}$$

intersect object

$$\frac{s_1.\texttt{type} = \texttt{object} \quad s_2.\texttt{type} = \texttt{object}}{\begin{array}{l}\{\texttt{allOf}:[s_1, s_2]\} \rightarrow \{\texttt{type}:\texttt{object}, \\ \quad \texttt{minProperties}:max(s_1.\texttt{minProperties}, s_2.\texttt{minProperties}), \\ \quad \texttt{maxProperties}:min(s_1.\texttt{maxProperties}, s_2.\texttt{maxProperties}), \\ \quad \texttt{required}:s_1.\texttt{required} \cup s_2.\texttt{required}, \\ \quad \texttt{patternProperties}:s_1.\texttt{patternProperties} \cup s_2.\texttt{patternProperties}\}\end{array}}$$

intersect anyOf

$$\frac{s_2 = \{\texttt{anyOf}:[s_{2_1}, \ldots, s_{2_n}]\}}{\{\texttt{allOf}:[s_1, s_2]\} \rightarrow \{\texttt{anyOf}:[\{\texttt{allOf}:[s_1, s_{2_1}]\}, \ldots, \{\texttt{allOf}:[s_1, s_{2_n}]\}]\}}$$

FIGURE 5.9: Simplification rules to eliminate `allOf` except for connective `not`.

$$\text{singleton anyOf}\ \frac{s.\texttt{anyOf} = [s_1]}{s \to s_1}$$

$$\text{fold anyOf}\ \frac{s.\texttt{anyOf} = [s_1, s_2, \ldots, s_n] \qquad n \geq 3}{s \to \{\texttt{anyOf}:[s_1, \{\texttt{anyOf}:[s_2, \ldots, s_n]\}]\}}$$

$$\text{union null}\ \frac{s_1.\texttt{type} = \texttt{null} \qquad s_2.\texttt{type} = \texttt{null}}{\{\texttt{anyOf}:[s_1, s_2]\} \to \{\texttt{type}:\texttt{null}\}}$$

$$\text{union boolean}\ \frac{s_1.\texttt{type} = \texttt{boolean} \qquad s_2.\texttt{type} = \texttt{boolean}}{\{\texttt{anyOf}:[s_1, s_2]\} \to \{\texttt{type}:\texttt{boolean}, \texttt{enum}:s_1.\texttt{enum} \cup s_2.\texttt{enum}\}}$$

$$\text{union string}\ \frac{s_1.\texttt{type} = \texttt{string} \qquad s_2.\texttt{type} = \texttt{string}}{\{\texttt{anyOf}:[s_1, s_2]\} \to \{\texttt{type}:\texttt{string}, \texttt{pattern}:s_1.\texttt{pattern} \cup s_2.\texttt{pattern}\}}$$

$$\text{union number}\ \frac{\begin{array}{c} s_1.\texttt{type} = \texttt{number} \qquad s_2.\texttt{type} = \texttt{number} \\ r_1 = \mathit{schema2range}(s_1) \qquad r_2 = \mathit{schema2range}(s_2) \\ r_1 \cap r_2 \neq \emptyset \end{array}}{\begin{array}{l} \{\texttt{anyOf}:[s_1, s_2]\} \to \{\texttt{anyOf}:[\\ \qquad \mathit{range2schema}(r_1 \cap r_2) \cup \{\texttt{multipleOf}:\mathit{gcd}(s_1.\texttt{multipleOf}, s_2.\texttt{multipleOf})\}, \\ \qquad \mathit{range2schema}(r_1 \setminus r_2) \cup \{\texttt{multipleOf}:s_1.\texttt{multipleOf}\}, \\ \qquad \mathit{range2schema}(r_2 \setminus r_1) \cup \{\texttt{multipleOf}:s_2.\texttt{multipleOf}\}]\} \end{array}}$$

FIGURE 5.10: Eliminating overlapping `anyOf`, except for connectives `not` and `allOf`, and types `array` and `object`.

5.3.3 *JSON Subschema Checking*

Given two canonicalized and simplified schemas, the third step of our approach checks whether one schema is a subtype of the other. Figure 5.11 presents inference rules defining the subschema relation on canonical, simplified schemas. All rules are algorithmically checkable, and all rules except for *schema uninhabited* are type-directed. To simplify their presentation, some of the rules use quantifiers, but all quantifiers are bounded and can thus be checked via loops.

Rule *schema uninhabited* states that an uninhabited schema is a subtype of any other schema. It uses an auxiliary *inhabited* predicate, which is elided for space but easily computable for primitives (recall that emptiness is decidable for regular languages). For structures, the predicate ensures that the schemas of all required components are inhabited. For logic connectives, the predicate is more involved but decidable. The rule for uninhabited types is the only rule that is not type-directed. Because canonicalization generally separates schemas by type, all other rules check same-typed schemas. We can handle uninhabited schemas independently of their type because there is no actual data of that type that would require type-specific reasoning.

Rule *subschema non-overlapping anyOf* handles `anyOf` schemas for the cases where simplification eliminates overlapping unions. Helper function *nonOverlapping* checks for unions of arrays and objects and conservatively assumes that those might overlap. In the non-overlapping case, it suffices to check the component schemas independently. For each schema on the left, we require a same-typed super schema on the right.

Rule *subschema number* is the most complicated of the subtype rules for primitive types due to `multipleOf` constraints. The simplifier cannot push negation through `multipleOf` constraints, and it cannot combine `allOf` combinations of such negated schemas. As a result, the rule has to handle multiple such constraints on both sides of the relation, with or without negation. We treat simple number schemas as single-element `allOf`s for consistency. This rule verifies that any number allowed by the set of constraints on the left is also allowed by the set of constraints on the right using an auxiliary *subNumber* relation, which is sketched in the following.

$$\text{uninhabited}\ \frac{\neg\, \mathit{inhabited}(s_1)}{s_1 <: s_2} \qquad \text{null}\ \frac{s_1.\texttt{type} = \texttt{null} \quad s_2.\texttt{type} = \texttt{null}}{s_1 <: s_2}$$

$$\text{boolean}\ \frac{s_1.\texttt{type} = \texttt{boolean} \quad s_2.\texttt{type} = \texttt{boolean} \quad s_1.\texttt{enum} \subseteq s_2.\texttt{enum}}{s_1 <: s_2}$$

$$\text{string}\ \frac{s_1.\texttt{type} = \texttt{string} \quad s_2.\texttt{type} = \texttt{string} \quad s_1.\texttt{pattern} \subseteq s_2.\texttt{pattern}}{s_1 <: s_2}$$

$$\text{number}\ \frac{\begin{array}{c}\forall i \in \{1,\ldots,k\}, \texttt{not} \notin \mathit{dom}(s_i) \wedge s_i.\texttt{type} = \texttt{number} \\ \forall i \in \{k+1,\ldots,n\}, \texttt{not} \in \mathit{dom}(s_i) \wedge s_i.\texttt{not.type} = \texttt{number} \\ \forall i \in \{1,\ldots,l\}, \texttt{not} \notin \mathit{dom}(t_i) \wedge t_i.\texttt{type} = \texttt{number} \\ \forall i \in \{l+1,\ldots,m\}, \texttt{not} \in \mathit{dom}(t_i) \wedge t_i.\texttt{not.type} = \texttt{number} \\ \mathit{subNumber}(([s_1,\ldots,s_k],[s_{k+1},\ldots,s_n]),([t_1,\ldots,t_l],[t_{l+1},\ldots,t_m]))\end{array}}{\{\texttt{allOf}:[s_1,\ldots,s_k,s_{k+1},\ldots,s_n]\} <: \{\texttt{allOf}:[t_1,\ldots,t_l,t_{l+1},\ldots,t_m]\}}$$

FIGURE 5.11: JSON Schema subtype inference rules.

(Continued on next page)

$$\text{array}\ \frac{\begin{array}{c} s_1.\texttt{type} = \texttt{array} \qquad s_2.\texttt{type} = \texttt{array} \\ s_1.\texttt{minItems} \geq s_2.\texttt{minItems} \qquad s_1.\texttt{maxItems} \leq s_2.\texttt{maxItems} \\ s_1.\texttt{items} = [s_{1_1}, \ldots, s_{1_k}] \qquad s_2.\texttt{items} = [s_{2_1}, \ldots, s_{2_m}] \\ \forall i \in \{0, \ldots, max(k,m)+1\},\ s_{1_i} \parallel s_1.\texttt{additionalItems} <: s_{2_i} \parallel s_2.\texttt{additionalItems} \\ s_2.\texttt{uniqueItems} \implies (s_1.\texttt{uniqueItems} \vee allDisjointItems(s_1)) \end{array}}{s_1 <: s_2}$$

$$\text{object}\ \frac{\begin{array}{c} s_1.\texttt{type} = \texttt{object} \qquad s_2.\texttt{type} = \texttt{object} \\ s_1.\texttt{minProperties} \geq s_2.\texttt{minProperties} \qquad s_1.\texttt{maxProperties} \leq s_2.\texttt{maxProperties} \\ s_1.\texttt{required} \supseteq s_2.\texttt{required} \\ \forall p_1 : s_{p_1} \in s_1.\texttt{patternProperties}, p_2 : s_{p_2} \in s_2.\texttt{patternProperties}, p_1 \cap p_2 \neq \emptyset \implies s_{p_1} <: s_{p_2} \end{array}}{s_1 <: s_2}$$

$$\text{non-overlapping anyOf}\ \frac{\forall i \in \{1, \ldots, n\}, \exists j \in \{1, \ldots, m\}, s_i <: t_j \qquad nonOverlapping([t_1, \ldots, t_m])}{\{\texttt{anyOf}: [s_1, \ldots, s_n]\} <: \{\texttt{anyOf}: [t_1, \ldots, t_m]\}}$$

FIGURE 5.11: JSON Schema subtype inference rules.

The *subNumber* relation first normalizes all schema range bounds by rounding them to the nearest included number that satisfies its `multipleOf` constraint. For each side, it then finds the least and greatest finite bound used. Every unbounded schema is split into two (or three for totally unbounded) schemas: one (or two) that are unbounded on one side, with the least/greatest bound as the other bound. The "middle" part is bounded. All these derived schemas keep the original `multipleOf`. The bounded schemas can all be checked (exhaustively if needed). For the unbounded schemas, we can separately check the positive and negative schemas, since they do not interact in interesting ways over unbounded sets. If PL and PR are the left and right positive schemas, and NL and NR are the left and right negative schemas, we verify that the constraints divide each other:

$$\forall_{pl \in PL}, \exists_{pr \in PR},\ pl.\texttt{multipleOf} \bmod pr.\texttt{multipleOf} = 0$$
$$\forall_{nr \in NR}, \exists_{nl \in NL},\ nr.\texttt{multipleOf} \bmod nl.\texttt{multipleOf} = 0$$

For example, because 3 divides 9 and 2 divides 4, we have:

```
{'allOf':[                          {'allOf':[
   {'type': 'number',                  {'type': 'number',
    'multipleOf': 9},          <:       'multipleOf': 3},
   {'type': 'number',                  {'type': 'number',
    'not': {'multipleOf': 2}}]}         'not': {'multipleOf': 4}}]}
```

Rule *subschema array* checks two array schemas. The left array size bounds should be within the size bounds of the right array. Additionally, the schema of every item specified in the former needs to be a subschema of the corresponding specification in the latter. If a schema is not explicitly provided, the schema provided by `additionalItems` is used. Recall that canonicalization adds in a default `additionalItems` schema if it was not specified. Additionally, if the right side specifies that the items must be unique, then the left needs to either specify the same or implicitly enforce this. For example,

```
{'type': 'array',                          {'type': 'array',
 'items': [{'enum': [0]}, {'enum': [1]}]}   <:   'uniqueItems': true]}
```

The *allDisjointItems* predicate checks for this by first obtaining the set of all the effective item schemas: every item schema for an index within the specified min/max bounds, and `additionalItems` if any allowed indices are unspecified. It then verifies that the conjunction of all pairs of effective items schemas are uninhabited.

Rule *subschema object* checks two object schemas. It first verifies that the number of properties of both sides have the appropriate relation, and that

the left side requires all the keys that the right side requires. Next, for every regular expression pattern p_1 on the left, if there is an overlapping regular expression pattern p_2 on the right, it checks that the corresponding schemas are subschemas. This check can be done separately for one pattern at a time thanks to the fact that canonicalization eliminates overlapping pattern properties.

5.4 IMPLEMENTATION

We implemented our subschema checker as an open-source Python library.[2] The implementation builds upon the jsonschema library[3] to validate schemas before running our subtype checking, the greenery library[4] for computing intersections of regular expressions, and the jsonref library[5] for resolving JSON schema references.

5.5 EVALUATION

This section evaluates our JSON subschema checker, which we refer to as jsonSubSchema. It addresses the following research questions:

RQ_1 How effective is jsonSubSchema in finding real bugs?

RQ_2 How correct and complete is jsonSubSchema?

RQ_3 How does our approach compare against prior work?

RQ_4 How efficient is jsonSubSchema?

5.5.1 *Experimental Setup*

We evaluate our approach on four datasets of pairs of JSON schemas listed in Table 5.2. The datasets cover different application domains and different ways of using JSON schemas. Snowplow is a service for data collection and aggregation for live-streaming of event data [Snob]. The service aggregates data from various sources and uses JSON schemas to specify data models, configurations of several components, etc. [Sno14]. We apply jsonSubSchema to 112 pairs of schemas that have consecutive

2 https://github.com/IBM/jsonsubschema
3 https://github.com/Julian/jsonschema
4 https://github.com/qntm/greenery
5 https://github.com/gazpachoking/jsonref

TABLE 5.2: Dataset details.

Project	Description	Schemas	Versions	Schema Pairs
Snowplow [Sno14]	Data collection & live-streaming of events	290	361	112
Lale [Lal]	Automated machine learning library	1,444	*NA*	2,818+28
Washington Post [Wp2]	Content creation and management	2,604	28	2,060
Kubernetes [Kuba]	Containerized applications	86,461	124	6,460
Total				11,478

versions and check whether the schema evolution is in line with semantic versioning. Lale is a Python library for type-driven automated machine learning [Lal]. It uses JSON schemas to describe the inputs and outputs of ML operators, as well as standard ML datasets. We apply jsonSubSchema to 2,818 schema pairs to find type errors in ML pipelines, and additionally, to 28 schema pairs with a known subtype relationship to validate the correctness of our approach. The Washington Post dataset is a collection of schemas describing content used by the Washington Post within the *Arc Publishing* content creation and management system [Wp2]. The final dataset comprises JSON schemas extracted from the OpenAPI specifications for Kubernetes [Kuba], a system for automating deployment, scaling, and management of containerized applications [Kubb]. For the last two datasets, we apply jsonSubSchema across each pair of consecutive versions of the same schema that introduces some change, to spot whether the change may impact the compatibility of the corresponding systems.

The total number of schema pairs is 11,478. Many of the schemas are of non-trivial size, with an average of 56KB and a maximum of 1,047KB. We use the first two datasets to evaluate the bug detection abilities of our approach (RQ_1), the last three datasets to evaluate the correctness, completeness (RQ_2 and RQ_3), and efficiency (RQ_4). To validate the correctness of the canonicalization and simplification steps of jsonSubSchema, we also use the official test suite for JSON Schema draft-04 [Jsoc]. It contains 146 schemas

and 531 JSON data instances that fully cover the JSON Schema language. All experiments used Ubuntu 18.04 (64-bit) on an Intel Core i7-4600U (2.10GHz) machine with 12GB RAM.

5.5.2 *RQ_1: Effectiveness in Detecting Bugs*

To evaluate the usefulness of jsonSubSchema for detecting bugs, we consider two real-world usage scenarios where the correctness of some software requires a specific subschema relation to hold. Overall, the approach detects 43 bugs, most of which are already fixed.

SCHEMA EVOLUTION BUGS IN SNOWPLOW Snowplow maintains versioned schemas that specify data sources and configurations while the system evolves [Sno14]. To ensure backward compatibility of clients and to avoid unnecessary updates of client code, the project adopts semantic versioning [Sem] to schemas using version numbers of the form *major.minor.patch* [Snoa]. For each schema evolution, we check whether the way a schema evolves is consistent with the way the version number changes. Specifically, a backward compatible schema fix corresponds to a patch increment, a change that adds functionality in a backward-compatible way to a minor increment, and a change that breaks backward compatibility to a major increment.

Our approach detects five schema evolution bugs, summarized in the top part of Table 5.3. We summarize multiple similar bugs into a single one for space reasons. For example, in *Snow-1* (Figure 5.12), two object properties changed their names. This change breaks backward compatibility for old data, hence, the major version number should have been incremented. The developers of Snowplow confirmed all reported bugs in Table 5.3 and acknowledged that specifically *Snow-1* and *Snow-2* are severe and require immediate attention. One developer wrote that "Our long-term plan is to implement an automatic algorithm to recognize versioning and clarify/formalise specification for known corner-cases". Our approach provides this kind of tool, and could help avoid schema evolution bugs in the future.

TABLE 5.3: 43 real-world bugs detected by jsonSubSchema.

Bug Id	Description	Bug Report Status
	Latent bugs in Iglu Central Snowplow schema versions	
Snow-1	*Breaking change.* Wrong increment of Patch version; should increment Major version instead.	Confirmed
Snow-2	*Wrong version.* Wrong update of Minor and Patch versions; should increment Patch instead.	Confirmed
Snow-3	*Spurious version increment.* Increment of Major version; should increment Patch instead.	Confirmed
Snow-4–5	*Spurious version increment.* Increment of Minor version; should increment Patch instead.	Confirmed
	Schemas of machine learning operators in Lale	
Lale-1–2	Classifiers output either string or numeric labels; output schemas should be union of arrays not array of unions.	Fixed
Lale-3–14	Classifiers should allow output labels to be strings or numeric instead of numeric only.	Fixed
Lale-15–32	Classifiers should allow output labels to be Booleans, beside strings or numeric.	Fixed
Lale-33	Using the empty schema in a union is too permissive and implies the empty schema, which validates everything.	Fixed
Lale-34–38	Using the empty schema as an output schema causes problems when used as input to the next operator.	Fixed

```
{'properties': {
   'event': {'type': 'object'},
   'error': {'type': 'string'},
   ...}
 'required': ['event', 'error'],
 'additionalProperties': false}
```

Version 1.0.0

```
{'properties': {
   'payload': {'type': 'object'},
   'failure': {'type': 'string'},
    ...}
 'required': ['payload', 'failure'],
 'additionalProperties': false}
```

Version 1.0.1

FIGURE 5.12: *Snow-1:* A schema evolution bug in Snowplow

```
{'type': 'array',
 'items': {
  'anyOf':[
   {'type': 'number'},
   {'type': 'string'}]}}
```

Wrong schema

```
{'anyOf': [
  {'type': 'array',
   'items': {'type': 'number'}},
  {'type': 'array',
   'items': {'type': 'string'}}]}
```

Correct schema

FIGURE 5.13: *Lale-1:* Wrong schema for an ML operator in Lale

INCORRECT ML PIPELINES IN LALE As a second real-world usage scenario, we apply jsonSubSchema to check interfaces in Lale [Hir+19] machine-learning pipelines before running those pipelines. Our approach detects 38 bugs that we summarize in the lower part of Table 5.3. All bugs have been fixed in response to finding them with jsonSubSchema.

For example, in *Lale-1* (Figure 5.13), a subschema check reveals that the output of several classifiers could either be numeric or string labels, but no inter-mixing of the two kinds. When introducing new operators into an ML pipeline, checking for such mistakes can save hours of training time for ML pipelines that fail due to incompatible input-output data, and prevent running a pipeline with incorrect data.

5.5.3 *RQ_2: Correctness and Completeness*

Given the complexity of JSON Schema, ensuring the correctness of our subschema checker is non-trivial. We aim for soundness, giving a correct answer whenever not returning "unknown", while being as complete as possible, i.e., trying to cover as many JSON Schema features as possible. The following evaluates to what extent our approach achieves this.

CANONICALIZATION AND SIMPLIFICATION Together, canonicalization and simplification aim at transforming a given schema into a simpler yet semantically equivalent schema. To check this property, we use the official JSON Schema test suite to validate the JSON instances in the suite against their corresponding schemas before and after the two transformations steps. Specifically, for schema s and its associated JSON data d in the JSON Schema test suite, we check whether:

$$\forall s, \forall d, \quad valid(d, s) \Leftrightarrow valid\big(d, simple(canonical(s))\big)$$

In all cases except one where jsonSubSchema yields a canonicalized schema, this new schema passes all relevant tests in the JSON schema test suite. This single case is due to an ambiguity of the specification of JSON schema and hence a mismatch between our own interpretation and the interpretation of the JSON schema validator of the semantics of the `allOf` connector when combined with the `additionalProperties` object constraint.

CORRECTNESS OF SUBTYPE CHECKING To evaluate the correctness of jsonSubSchema, we compare its results against a ground truth. Specifically, we gather pairs of schemas, along with their expected subtype relationship, in three ways. First, we randomly sample pairs that are textually different and manually assess their subtype relationship. Second, we sample consecutive versions of schemas from the Washington Post and Kubernetes datasets, and then manually assess their subtype relationship. Third, we gather 28 schema pairs from Lale, where the ground truth is whether or not the corresponding ML operator throws an exception when training on the corresponding dataset. By checking the schemas statically, our subtype checker can avoid such runtime errors.

In total, we gather 298 pairs with a ground truth subtype relationship, and our approach produces correct results for all of them, as summarized in Table 5.4. The $<:$, $:>$, $\equiv$, and $\not\equiv$ symbols represent the test performed on each pair. For example, for each pair $\langle s, t \rangle$ in the $<:$ row, the ground truth indicates whether $s <: t$ holds (positive, P) or not (negative, N). The jsonSubSchema part of the table shows the results of applying our subschema checker to each pair. The TP, TN, FP, and FN columns show the true positives, true negatives, false positives, and false negatives, respectively. For example, TN means that the tool produces the correct result (T for true) and that the ground truth indicates that the relationship being tested should not hold (N for negative).

TABLE 5.4: Effectiveness of jsonSubSchema and comparison to the existing isSubset tool.

		jsonSubSchema					isSubset				
	Pairs	Fail	TP	TN	FP	FN	Fail	TP	TN	FP	FN
$<:$	35	0	29	6	0	0	10	9	0	6	10
$:>$	35	0	31	4	0	0	10	21	0	4	0
$\equiv$	100	0	100	0	0	0	50	27	0	0	23
$\not\equiv$	100	0	63	37	0	0	0	63	0	37	0
Lale	28	0	12	16	0	0	7	3	10	0	8
Total	298	0	235	63	0	0	77	123	10	47	41

COMPLETENESS OF SUBTYPE CHECKING As mentioned in Section 5.3, there are some cases that jsonSubSchema does not decide. The following evaluates to what extent the approach covers the JSON Schema features that occur in real-world schemas. To this end, we apply the approach to 8,548 schemas pairs and count how often it refuses to give an answer.

Table 5.5 shows the cases when jsonSubSchema fails on our dataset. The subschema check fails for 4.24% of the pairs. The table shows three kinds of failures that happen in practice due to limitations of our approach. The first and most dominant failure reason is recursive schemas. This is not an inherent limitation of jsonSubSchema, but could be addressed by improving our implementation. The second case is negating object schemas. As seen in Table 5.5, only 0.34% of schema pairs fail due to the absence of this feature. In fact, the original schemas do not use negated schemas at all; instead, they were introduced as part of jsonSubSchema canonicalization (Section 5.3.1) where `oneOf` constraints are re-written into disjunction of conjunctions with negations. As mentioned in Section 5.3, JSON schema is not closed under union or negation of object schemas. The third case is when string or object schemas use "regular expressions" with non-regular extensions for textual patterns using the keywords `pattern` and `patternProperties`, respectively. Inclusion in non-regular languages (e.g., regular expressions with positive and negative look-around) is undecidable and beyond our scope.

TABLE 5.5: Reasons for incompleteness in practice.

Failure reason	Count	%
Recursive `$ref`	453	3.95%
Negated object schema	29	0.25%
Non-regular regex pattern	5	0.04%
Total	487	4.24%

5.5.4 *RQ_3: Comparison to Existing Work*

To our knowledge, we are the first to address the problem of JSON subschema checking for a large subset of JSON Schema, and there is no academic work that we can compare against. The closest developer tool we could find is isSubset [Hag19], a tool that states the same goal as ours: "Given a schema defining the output of some process A, and a second schema defining the input of some process B, will the output from A be valid input for process B?" We use the most recent version (1.0.6).

We run isSubset on the schema pairs with ground truth from Section 5.5.3. The right part of Table 5.4 shows the results. isSubset produces a non-negligible number of true positives, which means it indeed captures some of the semantic of the subtyping relation. However, isSubset also produces 47 false positives and 41 false negatives, i.e., gives a wrong answer to a subtype query. Overall, the existing tool gives a wrong answer in 40% of the cases where the tool does not fail.

To better understand isSubset's limitations, we inspected their code and tested it on simple schemas. Although the tool performs some simple semantic checks, e.g., it correctly reports `{'type':'integer'} <: {'type':'number'}`, it lacks the ability to capture the richness of JSON schema in many ways. For instance, it is oblivious to uninhabited schemas like `{'type':'string', 'enum':[1]}`, and it fails to detect that `{'type':['string','null']}` is equivalent to `{'type':['null', 'string']}`.

5.5.5 *RQ_4: Efficiency*

To evaluate how fast our subschema checker is in practice, we measure the time taken by subschema checks on a sample of 798 pairs of non-equal schemas from Table 5.2. We took every time measurement ten times and

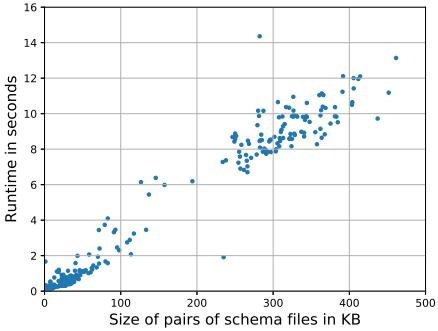


FIGURE 5.14: Efficiency of subschema checking.

report the average. Figure 5.14 shows the size of pairs of schema files in KB against the time subschema checking takes in seconds.

In most cases, our subschema checker terminates within a few seconds for moderately sized schemas, with time increasing roughly linearly with the schema file size. However, our subschema approach is lazy and terminates on the first violation of a subtyping rule. On one pair of schemas in our dataset, eliminated from the figure for scaling sake, it took around 2.8 minutes to terminate, which is not optimal for production. We will explore how to improve on this, for instance, by on-demand canonicalization.

5.6 CONTRIBUTIONS AND CONCLUSIONS

JSON schemas serve as documentation for their corresponding APIs. This chapter, therefore, supplements the thesis of this dissertation by leveraging JSON schemas to build a subtype checker, which eventually detects a new class of data-related bugs.

Our jsonSubSchema addresses a class of data compatibility bugs in applications that describe their data using JSON schemas, a class of bugs beyond the reach of classical static bug detectors (Chapter 2). The core of the approach is a novel subtype checker for such schemas. It addresses the various language features of JSON Schema by first canonicalizing and simplifying schemas, and by then type checking pairs of schema fragments that each describe data of a single type. The subtype checker successfully answers the subtype question for 96% of schemas that occur in the wild, clearly outperforming the best existing work. Applying the approach to

detect data compatibility bugs in popular web APIs and ML pipelines reveals 43 previously unknown bugs, most of which have already been fixed. We envision our work to contribute to more reliable software in data-intensive applications across different domains.

In summary, this chapter makes the following contributions:

- Formulating the problem of detecting data compatibility bugs as JSON subschema checking.
- A canonicalizer and simplifier that converts a given schema into a schema that is simpler to check yet permits the same set of documents.
- A subschema checker for canonicalized JSON schemas that uses separate subschema checking rules for each basic JSON type.
- Empirical evidence that the approach outperforms the state of the art, and that it reveals real-world data compatibility bugs in different domains.

6

NEURAL BUG-FINDING: A FUTURISTIC OUTLOOK

Static analysis has been used for decades to find software bugs (Chapter 2). Recent work [PS18; Vas+19; Wan+19; WCB14] shows that learning neural bug detection models from code examples could be a viable alternative to manually designing and implementing a program analysis. So far, these learning-based bug detectors have been most successful for bugs that are particularly hard to find for traditional analyses, e.g., name-related bugs or mismatches between comments and code. In contrast, it is unclear *how learned bug detectors compare with traditional program analysis in a head-to-head comparison.* In this chapter, we present *neural bug finding*, the first study to examine the effectiveness and challenges of using machine learning for general purpose static bug detection. Neural bug finding, as presented in this chapter, has the potential and advantage of utilizing the NL channel in source code, which is often neglected by classical bug detection approaches and the benefit of learning a bug detector end-to-end from data, without the need for coding sophisticated analysis frameworks. We perform a direct comparison of neurally learned and traditionally programmed bug detectors for 20 recurring bug patterns and draw on several conclusions and challenges for future work.

6.1 MOTIVATION

Static bug detectors find software bugs early during the development process by searching a code base for instances of common bug patterns. These tools, which we here call *bug detectors*, often consist of a scalable static analysis framework and an extensible set of checkers that each search for instances of a specific bug pattern. Examples of bug detectors include the

pioneering FindBugs tool [HP04], its successor SpotBugs[1], Google's Error Prone tool [Aft+12], and the Infer tool by Facebook [Cal+15].

Despite the overall success of static bug detection tools, there still remains a lot of potential for improvement. A recent study that applied state-of-the-art bug detectors to a set of almost 600 real-world bugs shows that over 95% of the bugs are currently missed [HP18a]. The main reason is that various different bug patterns exist, each of which needs a different bug detector. These bug detectors must be manually created, typically by program analysis experts, and they require significant fine-tuning to find actual bugs without overwhelming developers with spurious warnings. Bug detectors often require hundreds of lines of code each, even for bug patterns that seem trivial to find at first sight and when being built on top of a static analysis framework.

Recent work proposes to learn bug detectors from data instead of manually designing and implementing them [Har+18; Li+19; Li+18; PS18; Wan+19]. The main idea is to train a classifier that distinguishes between buggy and correct code examples, and to then query this classifier with previously unseen code. We call this line of work *neural bug finding* because these approaches reason about source code using deep neural networks. Neural bug finding has been shown to be successful for kinds of bugs that are hard to find for traditional bug detectors, e.g., bugs related to identifier names [PS18] or security vulnerabilities [Li+18]. However, these approaches are either designed and tuned for specific bug kinds [Li+18; PS18; Wan+19] or intended for general defect prediction, i.e., they predict only that some code fragment is buggy, but not what kind of bug it is [Har+18; Li+19]. In contrast, it is unclear how learned bug detectors compare with traditional program analyses in a head-to-head comparison. Learning bug detectors from source code examples has the potential of utilizing the dual modality of source code: (i) The algorithmic channel expressed through the syntax and structure of the code and (ii) The NL modality embedded in identifiers and types names, often picked by programmers to express their intentions and understanding. Specifically, the NL modality is a new dimension to exploit for general purpose bug detection, which is often under-utilized by traditional bug detectors.

This chapter presents an empirical study of the opportunities and challenges in neural bug detection. The study compares a set of fully automatically learned bug detector with more traditionally created, static

1 https://spotbugs.github.io

analysis-based bug detectors that are used in practice. We address the following questions:

- Does the effectiveness of neural bug finding come close to the existing, manually created bug detectors?
- When and why does neural bug detection work?
- How do properties of the training and validation, e.g., their size and composition, influence neural bug detection?
- What pitfalls do exist when evaluating neural bug finding models?

Answering these questions will help assess the opportunities provided by neural bug detection and will guide future work to address relevant open challenges.

To study learned bug detectors in a head-to-head comparison with traditional bug detection, we use an existing, traditionally developed bug detector as a generator of training data. To this end, we run the existing bug detector on a corpus of code to obtain warnings about specific kinds of bugs. Using these warnings and their absence as a ground truth, we then train a neural model to distinguish code with a particular kind of warning from code without such a warning. For example, we train a model that predicts whether a piece of code uses reference equality instead of value equality for comparing objects in Java. This setup allows us to assess to what extent neural bug finding can imitate existing bug detectors.

One potential drawback of using an existing bug detector as the data generator is that some warnings may be spurious and that some bugs may be missed. To mitigate this problem, we focus on bugs flagged by bug detectors that are enabled in production in a major company and that empirically show false positive rates below 10% [Sad+15]. Another drawback is that the learned bug detectors are unlikely to outperform the static analyzers they learn from. However, the purpose of this work is to study whether training a model for neural bug finding is feasible, whereas we leave the problem of obtaining training data beyond existing static analyzers as future work.

6.2 METHODOLOGY

Our approach applies machine learning (ML), specifically deep learning, to source code and learns a model that predicts whether a given piece

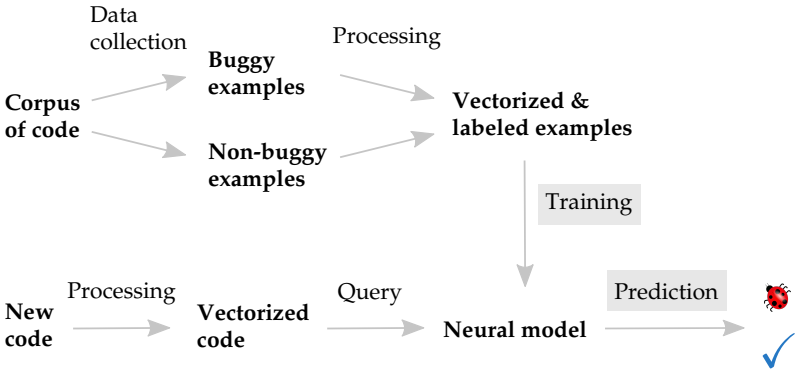


FIGURE 6.1: Overview of neural bug finding.

of code suffers from a specific bug or not. Figure 6.1 gives an overview of the neural bug finding approach. As training and validation data, we gather hundreds of thousands of code examples, some of which are known to contain specific kinds of bugs, from a corpus of real-world software projects (Section 6.2.1). To feed these code examples into a neural model, we abstract and vectorize the source code of individual methods (Section 6.2.2). A particularly interesting challenge is how to select examples of buggy and non-buggy code for training the bug detection model, which we address in Section 6.2.3. Finally, Section 6.2.4 describes how we train recurrent neural network (RNN) models that predict different kinds of bugs.

6.2.1 *Gathering Data*

To study the capability of neural bug finding, we need some kind of *oracle* that provides examples of buggy and non-buggy code to train ML models. One could potentially collect such data from existing bug benchmarks [Cif+09; JJE14; Lu+05; Tom+19]. Unfortunately, such bug benchmarks provide at most a few hundreds of buggy code examples, which is a relatively small number for training neural networks. Other directions include mining existing code repositories for pull requests and commits that fix bugs or generating training data by injecting bugs, e.g., via mutations.

In this work, we obtain examples of buggy and non-buggy code by running a state-of-the-art static analyzer as an oracle on a large corpus of code, and by collecting warnings produced by the static analyzer. We use Error Prone [Aft+12] as the oracle, a state-of-the-art static bug finding

tool for Java, which is developed and used by Google, and made available as open-source. We run Error Prone on the Qualitas Corpus [Tem+10], a curated set of 112 open-source Java projects and collect all warnings and errors reported by Error Prone along with their corresponding kinds and code locations. To simplify terminology, we call all problems reported by Error Prone a "bug", irrespective of whether a problem affects the correctness, maintainability, performance, etc. of code.

Table 6.1 shows the bug kinds we consider in this work. Error Prone warnings flag class-level problems, e.g., mutable `enums`; method-level problems, e.g., missing annotations, such as the `@Override` annotation (Id 1 in Table 6.1); and statement-level and expression-level issues, such as expressions with confusing operator precedence (Id 9 in Table 6.1). Since most of the warnings are at the method level or at the expression level, our study focuses on learning to predict those bugs, ignoring class-level bugs. After removing class-level bugs, Table 6.1 includes the 20 most common kinds of bugs reported by Error Prone on the Qualitas corpus.

To illustrate that finding these bugs with traditional means is non-trivial, the last column of Table 6.1 shows how many non-comment, non-empty lines of Java code each bug detector has. On average, each bug detector has 170 lines of code, in addition to th 156k lines of general infrastructure and test code in the Error Prone project. These numbers show that manually creating bug detectors is a non-trivial effort that would be worthwhile to complement with learned bug detectors.

6.2.2 *Representing Methods as Vectors*

6.2.2.1 *Code as Token Sequences*

The next step is to model source code in a manner that enables us to apply machine learning to it to learn patterns of buggy and non-buggy code. Among the different approaches, we here choose to represent code as a sequence of tokens. This representation is similar to natural languages [Col+11; Hin+12] and has seen various applications in programming and software engineering tasks, such as bug detection [Wan+16], program repair [BS16; Gup+17], and code completion [RVY14].

TABLE 6.1: Top 20 warnings reported by Error Prone on the Qualitas Corpus.

Id	Warning	Count	Description	LoC
1	MissingOverride	268,304	Expected `@Override` because method overrides method in supertype; including interfaces	111
2	BoxedPrimitiveConstructor	3,769	`valueOf` or autoboxing provides better time and space performance	268
3	SynchronizeOnNonFinalField	2,282	Synchronizing on non-final fields is not safe if the field is updated	66
4	ReferenceEquality	1,680	Compare reference types using reference equality instead of value equality	282
5	DefaultCharset	1,550	Implicit use of the platform default charset, can result in unexpected behavior	515
6	EqualsHashCode	590	Classes that override `equals` should also override `hashCode`	106
7	UnsynchronizedOverridesSynchronized	517	Thread-safe methods should not be overridden by methods that are not thread-safe	125
8	ClassNewInstance	486	`Class.newInstance()` bypasses exception checking	254
9	OperatorPrecedence	362	Ambiguous expressions due to unclear precedence	118
10	DoubleCheckedLocking	204	Double-checked locking on non-volatile fields is unsafe	305

(Continued on next page)

Table 6.1: Top 20 warnings reported by Error Prone on the Qualitas Corpus.

Id	Warning	Count	Description	LoC
11	NonOverridingEquals	165	A method that looks like `Object.equals` but does not actually override it	179
12	NarrowingCompoundAssignment	158	Compound assignments like `x += y` may hide dangerous casts	167
13	ShortCircuitBoolean	116	Prefer the short-circuiting boolean operators `&&` and `||` to `&` and `|`	88
14	IntLongMath	111	Expression of type `int` may overflow before being assigned to a `long`	127
15	NonAtomicVolatileUpdate	80	Update of a volatile variable is non-atomic	142
16	WaitNotInLoop	77	`Object.wait()` and `Condition.await()` must be called in a loop to avoid spurious wakeups	76
17	ArrayToString	56	Calling `toString` on an array does not provide useful information (prints its identity)	256
18	MissingCasesInEnumSwitch	53	Switches on `enum` types should either handle all values, or have a default case	86
19	TypeParameterUnusedInFormals	46	A method's type parameter is not referenced in the declaration of any of the formal parameters	135
20	FallThrough	45	`switch` case may fall through	96
	Total	280,651		3,402

Let M be the set of all non-abstract Java methods in our corpus of code. For each method $m \in M$, we extract the sequence of tokens s_m from the method body, starting at the method definition and up to length n. Let S be the set of all sequences extracted from all methods M. Extracted tokens include keywords such as `for`, `if`, and `void`; separators such as `;`, `()`, and `,`; identifiers such as variable, method, and class names; and finally, literals such as `5` and `"abc"`. Each token $t_i = (lex, t, l)$, where $1 \leq i \leq n$, is a tuple of the lexeme itself, its type t, and the line number l at which t occurs in the source file. We ignore comments. As a default, we choose a sequence length of $n = 50$ in our experiments.

As an alternative to a token sequence-based code representation, we could model code, e.g., as abstract syntax trees (ASTs), control-flow graphs (CFGs), or program-dependence graphs (PDGs). Recent work has started to explore the potential of graph-based code representations [ABK17; Alo+18a; Alo+18b; Bro+18; DTR18]. We here focus on a simpler, sequence-based code representation, because it has been found to provide a reasonable baseline and to avoid favoring any of the more specialized ways of modeling code.

6.2.2.2 *Representing Tokens*

To enable the ML model to learn and generalize patterns from source code, we abstract the extracted token sequences in such a way that discovered patterns are reusable across different pieces of code. One challenge is that source code has a huge vocabulary due to identifiers and literals chosen by developers [BJR19]. To mitigate this problem, we extract a vocabulary V consisting of the most frequent keywords, separators, identifiers, and literals from all code in our corpus. In addition to the tokens in the corpus, we include two special tokens: `UNK`, to represent any occurrence of a token beyond the most frequent tokens, and `PAD` to pad short sequences. In our experiments, we set $|V| = 1000$ which covers 82% of all keywords, separators, identifiers, and literals in our corpus.

We convert the sequences of tokens of a given code example to a real-valued vector by representing each token t through its one-hot encoding. The one-hot encoding function $H(t)$ returns a vector of length $|V|$, where all elements are zero except one that represents the specific token t. To allow the learned models to generalize across similar tokens, we furthermore learn an embedding function E that maps $H(t)$ to $\mathbb{R}^e$, where e is the embedding size. Based on these two functions, we represent a sequence of tokens $s \in S$ through a real-valued vector v_s as follows:

Definition 6.2.1 (Source Code Vector) *For a sequence of tokens $s \in S$ that is extracted from the source code of method $m \in M$, where the length of s is n and $s = t_1, t_2, \ldots, t_n$, the vector representation of s is*

$$v_s = [E(H(t_1)), E(H(t_2)), \ldots, E(H(t_n))]$$

6.2.3 *Buggy and Non-Buggy Examples*

The training and validation data consists of two kinds of code examples: buggy and non-buggy examples. We focus on methods as code examples, i.e., our neural bug detectors predict whether a method contains a particular kind of bug. Let K be the set of all bug kinds that the oracle can detect and W be the set of all warnings reported by it on the Qualitas corpus. Each warning $w \in W$ is represented as $w = (k, l, m)$ where $k \in K$ is the bug kind flagged at line number l in method m. For each kind of bug $k \in K$, we consider two subsets of M:

- The set $M_{k_{\text{bug}}}$ of methods flagged by the oracle to suffer from bug kind k.
- The set $M_{k_{\text{nBug}}}$ of methods for which the oracle does not report any bug of kind k.

Based on these two sets, we select a subset of the methods as examples to train and validate our models, as described in the following. After selecting the methods, we produce two sets of sequences, $S_{k_{\text{bug}}}$ and $S_{k_{\text{nBug}}}$, as described in Section 6.2.2.

6.2.3.1 *Selecting Non-Buggy Examples*

One strategy for selecting non-buggy examples is to randomly sample from all methods that are not flagged as buggy for a bug of kind k. However, we found this naive approach to bias the learned model towards the presence or absence of specific tokens related to k, but not necessarily sufficient to precisely detect k. For example, when training a model to predict a problem with binary expressions (Id 9 in Table 6.1), using the naive approach to select non-buggy examples would result in a model that learns to distinguish source code sequences that contain binary expressions from sequences that do not. In other words, it would simply flag any binary expression as potentially buggy.

To address this problem, we selectively pick non-buggy examples that are similar to the buggy examples, but that do not suffer from the same programming error k. For example, if a warning kind k flags binary expressions, we would like $S_{k_{\text{nBug}}}$ to be mostly composed of sequences that include binary expressions but that do not suffer from k. To select such similar examples in an automated manner, we perform two steps. First, we convert each sequence into a more compact vector that summarizes the tokens in the sequence. Second, we query all non-buggy examples for those similar to a given buggy example using a nearest neighbor algorithm. The following explains these two steps in more detail.

The first step converts sequence vectors to frequency vectors. Let $v_s = [t_1, t_2, \ldots, t_n]$ be a vector of n tokens corresponding to code sequence s. We convert v_s into a vector of frequencies v_s^{freq} of all words in V. In other words, we compute:

$$v_s^{\text{freq}} = [count(t_{i_1}, s), count(t_{i_2}, s), \ldots, count(t_{i_{|V|}}, s)]$$

for some fixed ordering $i_1, i_2, \ldots, i_{|V|}$ of the vocabulary V, and where $count(t, s)$ returns the number of occurrences of t in s. We exclude the special tokens `UNK` and `PAD` when computing v_s^{freq}.

Before searching the space of non-buggy examples using the token-frequency vectors, we counteract the effect of tokens with very high frequencies. Examples of these tokens include `new`, `=`, `return`, and separators, all of which are likely to appear across many different sequences of source code but are less relevant for selecting non-buggy examples. To counteract their influence, we apply *term frequency-inverse document frequency (TF-IDF)*, which offsets the number of occurrences of each token in the frequency vectors by the number of sequences this token appears in. TF-IDF is widely used in information retrieval and text mining to reflect how important a word is to a document in a corpus of documents, while accommodating for the fact that some words occur more frequently than others.

As the second step, to search the space of non-buggy code sequences in our data set, we use an efficient, high-dimensional search technique called approximated nearest neighbor (ANN). We use ANN to search the vector representations of all non-buggy methods for a subset $S_{k_{\text{nBug}}}^{\text{ANN}}$ of non-buggy examples that are similar to the multi-dimensional space of sequence vectors in $S_{k_{\text{bug}}}$.

Definition 6.2.2 (ANN Non-Buggy Examples) *For every buggy code example* $s_{k_{\text{bug}}} \in S_{k_{\text{bug}}}$ *of bug kind* $k \in K$*, the ANN of* $s_{k_{\text{bug}}}$ *is* $\text{ANN}_{\text{search}}(s_{k_{\text{bug}}}, S_{k_{\text{nBug}}})$

where $\text{ANN}_{\text{search}}(x, Y)$ *returns the ANN of* x *in* Y*. Therefore, the set of non-buggy nearest neighbors sequences of* $S_{k_{\text{bug}}}$ *is:*

$$S^{\text{ANN}}_{k_{\text{nBug}}} = \{ s' \in S_{k_{\text{nBug}}} \mid s' = \text{ANN}_{\text{search}}(s, S_{k_{\text{nBug}}}) \quad \forall s \in S_{k_{\text{bug}}} \}$$

ANN uses locality sensitive hashing to perform this high-dimensional space search, which is much more efficient than exhaustively computing pair-wise distances between all vectors.

6.2.3.2 *Selecting Buggy Examples*

When selecting sequences $S_{k_{\text{nBug}}}$ of non-buggy examples, we need to consider whether the location of the bug is within the first n tokens of the method. A warning $w = (k, l_w, m)$ that flags line l_w in method m could fall beyond the sequence s_m extracted from m if the last token of s_m, $t_n = (lex, t, l_{t_n})$ has $l_{t_n} < l_w$. In other words, it could be that a warning flagged at some method by the oracle occurs at a line beyond the extracted sequence of that method because we limit the sequence length to n tokens. In such a case, we remove this example from the set of buggy examples of bug kind k and we use it as a non-buggy example.

6.2.4 *Learning Bug Detection Models*

The remaining step in our neural bug finding approach is training the ML model. Based on the vector representation of buggy and non-buggy examples of code sequences, we formulate the bug finding problem as binary classification.

Definition 6.2.3 (Bug Finding Problem) *Given a previously unseen piece of code* C*, the problem* $P_k : C \to [0, 1]$ *is to predict the probability that* C *suffers from bug kind* k*, where* 0 *means certainly not buggy and* 1 *means that* C *certainly has a bug of kind* k*.*

We train a model to find a bug of kind k in a supervised setup based on two types of training examples: buggy examples (v_{bug}, 1) and non-buggy examples (v_{nBug}, 0), where v_{bug} and v_{nBug} are the vector representations of buggy and non-buggy code, respectively. During prediction, we interpret a predicted probability lower than 0.5 as "not buggy", and as "buggy" otherwise.

Since we model source code as a sequence of tokens, we employ recurrent neural networks (RNNs) as models. In particular, we use bi-directional RNN with Long Short Term Memory (LSTM) [GSC99] units. As the first layer, we have an embedding layer that reduces the one-hot encoding of tokens into a smaller embedding vector of length 50. For the RNN, we use one hidden bi-directional LSTM layer of size 50. We apply a dropout of 0.2 to the hidden layer to avoid overfitting. The final hidden states of the RNN are fed through a fully connected layer to an output layer of dimension 1, and the sigmoid activation function is applied to the output. For the loss function, we choose binary cross entropy, and we train the RNN using the Adam optimizer. Finally, we use a dynamically calculated batch size based on the size of the training data (10% of the size of the training set with a maximum of 300).

6.2.5 *Different Evaluation Settings*

We study four different ways of combining training and validation data, summarized in Table 6.2. These four ways are combinations of two variants of selecting code examples. On the one hand, we consider balanced data, i.e., with an equal number of buggy and non-buggy examples. On the other hand, we consider a stratified split, which maintains a distribution of buggy and non-buggy examples similar to that in all the collected data, allowing us to mimic the frequency of bugs in the real-world. For instance, assume the total number of samples collected for a specific warning kind is 200 samples, of which 50 (25%) are buggy and 150 (75%) are not buggy. If we train the model with 80% of the data and validate on the remaining 20%, then a stratified split means the training set has 160 samples, of which 40 (25%) are buggy and 120 (75%) are not buggy, and the validation set has 40 samples, of which 10 (25%) are buggy and 30 (75%) are not buggy.

Evaluation setups BS and $B_{ANN}S$ correspond to the scenario of using balanced data for training and stratified split for validation. In setup BS, we randomly sample the non-buggy examples to build a balanced training set, while in setup $B_{ANN}S$ we use our novel approximated nearest neighbor (ANN) search to find non-buggy examples (Section 6.2.3). Since for many of the kinds of warnings the number of collected buggy examples is relatively small for a deep learning task, we additionally evaluate a third setup, SS, where we utilize all non-buggy data available by doing a stratified split for training and validation. Finally, setup BB represents the most traditional

TABLE 6.2: Setups used to evaluate the neural bug finding models.

Experiment	Training	Validation
BS	Balanced	Stratified
$B_{ANN}S$	Balanced (ANN sampling)	Stratified
SS	Stratified	Stratified
BB	Balanced	Balanced

setup for binary classifiers, which uses balanced training and balanced validation sets.

6.3 IMPLEMENTATION

We use the JavaParser[2] to parse and tokenize all Java methods in the Qualitas corpus. Tokenized methods, warnings generated by Error Prone, their kinds, and locations are stored in JSON files for processing by the models. Python scikit-learn[3] is used to compute the TD-IDF of all examples and NearPy[4] is used to find the ANN of each buggy example. To implement the recurrent neural networks, we build upon Keras[5] and Tensorflow[6].

6.4 RESULTS

We study neural bug finding by posing the following research questions:

- RQ_1: How effective are neural models at identifying common kinds of programming errors?
- RQ_2: Why does neural bug finding sometimes work?
- RQ_3: Why does neural bug finding sometimes not work?
- RQ_4: How does the composition of the training data influence the effectiveness of a neural model?

2 http://javaparser.org
3 https://scikit-learn.org
4 http://pixelogik.github.io/NearPy
5 https://keras.io
6 https://www.tensorflow.org

- RQ_5: How does the amount of training data influence the effectiveness of a neural model?
- RQ_6: What pitfalls exist when evaluating neural bug finding?

6.4.1 *Experimental Setup*

For each experiment, we split all available data into 80% training data and 20% validation data, and we report the results with the validation set. Each experiment is repeated five times, and we report the average results. For the qualitative parts of our study, we systematically inspected at least ten, often many more, validation samples from each warning kind. All experiments are performed on a machine with 48 Intel Xeon E5-2650 CPU cores, 64GB of memory, and an NVIDIA Tesla P100 GPU.

6.4.2 *RQ_1: How effective are neural models at identifying common kinds of programming errors?*

To study the effectiveness of the neural bug finding models, we measure their precision, recall, and F1-score. For a specific bug kind, precision is the percentage of actual bugs among all methods that the model flags as buggy, and recall is the percentage of bugs detected by the model among all actual bugs. The F1-score is the harmonic mean of precision and recall.

We first look at Experiment $B_{ANN}S$, which uses balanced training data selected using ANN and an imbalanced validation set. The results of this and the other experiments are shown in Table 6.3. Across the 20 kinds of warnings we study, precision ranges between 73.5% down to 0.04%, while recall ranges between 97.76% and 43.6%. The relatively high recall shows that neural bug finders find a surprisingly high fraction of all bugs. However, as indicated by the low precision for many warnings kinds, the models also tend to report many spurious warnings.

Table 6.3: Precision, recall, and F1-score of the neural bug finding models of the top 20 warnings reported by Error Prone. Results are obtained by training with 80% of available data and validating on the remaining 20%. Table also shows the total number of examples available in the data set. Warnings are in descending order by their total number of buggy examples.

				Experiment BS			Experiment $B_{ANN}S$			Experiment SS			Experiment BB		
		Nb. of examples		Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1
Id	Warning kind	Buggy	nBuggy	%	%	%	%	%	%	%	%	%	%	%	%
1	MissingOverride [G]	268,304	501,937	69.74	86.05	76.97	73.53	77.70	75.48	79.78	74.97	77.28	82.34	84.25	83.24
2	BoxedPrimitiveConstructor [L]	3,769	767,112	12.00	96.47	21.23	17.47	93.93	29.20	93.62	92.02	92.67	95.51	94.26	94.85
3	Sync.OnNonFinalField [G]	2,282	653,856	20.19	98.73	33.18	24.14	97.76	38.57	71.05	79.43	74.88	96.28	99.74	97.97
4	ReferenceEquality [G]	1,680	746,285	1.48	89.17	2.90	1.55	83.21	3.05	78.94	39.40	52.08	85.01	90.40	87.51
5	DefaultCharset [G]	1,550	747,192	2.18	95.35	4.27	4.06	80.00	7.69	75.61	60.58	66.57	91.83	94.56	93.13
6	EqualsHashCode [G]	590	673,446	8.20	99.49	14.91	8.79	85.25	15.89	39.71	5.93	10.06	98.38	100.00	99.17
7	Unsync.OverridesSync. [G]	517	657,303	0.36	82.14	0.72	0.28	68.93	0.55	61.26	16.89	25.27	85.74	77.05	80.73
8	ClassNewInstance [G]	486	742,585	0.80	94.23	1.59	2.56	85.36	4.97	88.04	79.59	83.46	91.41	93.97	92.44
9	OperatorPrecedence [L]	362	716,691	0.51	92.22	1.02	0.49	75.56	0.98	70.10	20.28	30.00	89.91	88.67	89.17
10	DoubleCheckedLocking [G]	204	297,959	2.80	97.56	5.40	5.05	95.61	9.24	95.80	83.41	88.84	98.30	95.53	96.77

(Continued on next page)

TABLE 6.3: Precision, recall, and F1-score of the neural bug finding models of the top 20 warnings reported by Error Prone. Results are obtained by training with 80% of available data and validating on the remaining 20%. Table also shows the total number of examples available in the data set. Warnings are in descending order by their total number of buggy examples.

				Experiment BS			Experiment $B_{ANN}S$			Experiment SS			Experiment BB		
		Nb. of examples		Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1
Id	Warning kind	Buggy	nBuggy	%	%	%	%	%	%	%	%	%	%	%	%
		Nb. of examples		Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1	Pr.	Re.	F1
11	NonOverridingEquals [L]	165	488,094	2.04	93.33	3.94	2.97	77.58	5.61	90.01	87.88	88.63	95.22	97.95	96.49
12	NarrowingCompoundAssign. [L]	158	660,390	0.29	88.12	0.58	0.34	79.38	0.68	53.04	31.25	38.11	92.72	92.22	92.45
13	ShortCircuitBoolean [L]	116	616,037	0.09	82.61	0.18	0.10	73.91	0.20	72.22	31.30	39.70	78.21	91.82	83.78
14	IntLongMath [L]	111	531,502	0.23	79.09	0.47	0.30	81.82	0.59	59.52	7.27	12.60	90.82	100.00	94.95
15	NonAtomicVolatileUpdate [G]	80	369,501	0.07	71.25	0.15	0.04	71.25	0.08	0.00	0.00	0.00	80.24	83.60	81.00
16	WaitNotInLoop [G]	77	469,210	0.27	97.33	0.53	0.30	86.67	0.59	83.17	49.33	61.52	89.75	100.00	94.57
17	ArrayToString [L]	56	554,213	0.07	96.36	0.13	0.04	61.82	0.08	20.00	1.82	3.33	96.36	96.67	96.18
18	MissingCasesInEnumSwitch [G]	53	430,701	0.10	85.45	0.20	0.05	43.64	0.10	0.00	0.00	0.00	81.97	94.64	87.09
19	TypeParam.UnusedInFormals [L]	46	321,451	0.41	86.67	0.81	0.69	93.33	1.35	0.00	0.00	0.00	92.70	93.33	92.50
20	FallThrough [L]	45	615,140	0.08	93.33	0.15	0.43	82.22	0.84	63.33	20.00	30.09	83.44	92.29	87.13
	Median			0.46	92.78	0.92	0.59	80.91	1.17	70.58	31.28	38.91	91.12	94.12	92.48

In Experiment SS, we use a much larger, but imbalanced, training set. Table 6.3 also shows the results of this experiment. One can observe a clear improvement of precision over Experiment $B_{ANN}S$ for many of the models. This improvement in precision is due to the richer and larger training set, which trains the model with many more non-buggy examples than Experiment $B_{ANN}S$, making it more robust against false positives. However, the increased precision comes at the cost of decreasing recall compared to Experiment $B_{ANN}S$. For example, the neural model that predicts double checked locking bugs (Id 10 in Table 6.3) has its recall dropping from 95.6% to 83.4% when using the full training data available. Yet, the reduced recall is offset by a huge increase in precision, causing the median F1-score to grow from 1.17% in Experiment $B_{ANN}S$ to 38.91% in Experiment SS.

> The effectiveness of neural bug finders varies heavily across bug patterns, reaching precision and recall values above 90% for some bug patterns, but struggling to be on par with traditional bug detectors for many other patterns.

6.4.3 *RQ_2: Why does neural bug finding work?*

To answer this question and also RQ_3, we systematically inspect true positives, true negatives, false positives, and false negatives for each model. We discuss our observations by splitting the warning kinds into two groups, based on whether the information provided to the neural model is, in principle, sufficient to accurately detect the kind of bug.

6.4.3.1 *Bug Kinds with Sufficient Available Information*

The first group includes all bug kinds where the bug pattern could, in principle, be precisely and soundly detected based on the information we provide to the neural model. Recall that we feed the first 50 tokens of a method into the model, and no other information, such as the class hierarchy or other methods in the program. In other words, the model is given enough information to reason about local bugs, which involve a property of one or a few statements, one or a few expressions, or the method signature. We mark all warning kinds in this group with a $^{\text{L}}$ (for local) in Table 6.3. Intuitively, these warning kinds correspond to what traditional lint-like tools may detect based on a local static analysis.

We now discuss examples of true positives, i.e., correctly identified bugs, among the warnings reported by models trained for warning kinds in the first group.

BOXED PRIMITIVE CONSTRUCTOR (ID 2) This bug pattern includes any use of constructors of primitive wrappers, such as `new Integer(1)` and `new Boolean(false)`. The neural bug finder for this warning achieves high precision and recall of 93.6% and 92% respectively (Table 6.3, Experiment SS). The following is an instance of this bug, which is detected by the neural model:

```
public int compareTo(java.lang.Object o) {
 return new Integer(myX).compareTo(new Integer(((NodeDisplayInfo)o).myX)); }
```

Inspecting these and other bug kinds shows that, in essence, the model learns to identify specific subsequences of tokens, such as `new Boolean` and `new Integer`, as a strong signal for a bug.

OPERATOR PRECEDENCE (ID 9) This warning is about binary expressions that either involve ungrouped conditionals, such as `x || y && z`, or a combination of bit operators and arithmetic operators, such as `x + y << 2`. Such expressions are confusing to many developers and should be avoided or made more clear by adding parentheses. The following is a true positive detected by our neural model.

```
@Override
public int nextPosition() {
  assert (positions != null && nextPos < positions.length) || startOffsets != null &&
        nextPos < startOffsets.length;
  ... }
```

Overall, the neural model achieves 70% precision and 20.28% recall. The fact that the model is relatively successful shows that neural bug finders can learn to spot non-trivial syntactic patterns. Note that the space of buggy code examples for this warning kind is large, because developers may combine an arbitrary number of binary operators and operands in a single statement. Given that the model is trained on very few buggy examples, 290 (80% of 362), the achieved precision and recall are promising.

> The models learn syntactic patterns commonly correlated with particular kinds of bugs and identify specific tokens and token sequences, such as calls to particular APIs.

6.4.3.2 *Bug Kinds with Only Partial Information*

The second group of bug kinds contains bug patterns that, in principle, require more information than available in the token sequences we give to the neural models to be detected soundly and precisely. For example, detecting these kinds of bugs requires information about the class hierarchy or whether a field used in a method is final. We mark these bug kinds with a ^{G} (for global) in Table 6.3. The bug kinds include bugs that require type and inheritance information, e.g., missing override annotations (Id 1), missing cases in enum switch (Id 18), default Charset (Id 5), and unsynchronized method overriding a synchronized method (Id 7). They also include bugs for which some important information is available only outside the current method, such as synchronized on non-final field (Id 3) and equals-hashcode (Id 6). Note that although detecting these bugs requires information beyond the sequence of tokens extracted from the methods, the bug location lies within the sequence of tokens. Somewhat surprisingly, neural bug finding also works for some of these bug patterns, achieving precision and recall above 70% in some cases, which we describe in the following.

MISSING `@override` (ID 1) This warning is for methods that override a method of an ancestor class but that do not annotate the overriding methods with `@Override`. Although the supertype information that is required to accurately detect this problem is not available to the neural model, the model provides high precision and recall. Inspecting true positive predictions and training examples reveals that the model learns that many overriding methods override methods of common Java interfaces and base classes. Examples include the `toString()` method from the `Object` base class and the `run()` method from the `Runnable` interface. In fact, both method names appear in the data set as buggy 44,789 and 21,767 times, respectively. In other words, the models successfully learns to identify common instances of the bug pattern, without fully learning the underlying bug pattern.

DEFAULT `charset` (ID 5) This warning flags specific API usages that rely on the default Charset of the Java VM, which is discouraged for lack of portability. The "pattern" to learn here are specific API names, which implicitly use the default Charset. The following instance is a true positive detected by the neural model:

```
1 private void saveTraining() {
2  BufferedWriter writer = null;
3  try {
4   writer = new BufferedWriter(new FileWriter(SAVE_TRAINING));
5   ...
```

As we show in RQ_3, this bug is more subtle than it looks. Correctly detecting this problem requires, in some cases, information on the type of receiver objects, on which the APIs are called.

DOUBLE CHECKED LOCKING (ID 10) This bug is about a lazy initialization pattern [Laz] where an object is checked twice for nullness with synchronization in-between the null checks, to prevent other threads from initializing the object concurrently. The following is a true positive reported by our neural model.[7]

```
1 private SimpleName pointcutName = null;
2 ...
3 public SimpleName getName() {
4  if (this.pointcutName == null) {
5   synchronized (this) {
6    if (this.pointcutName == null) {
7     ...
8  return this.pointcutName; }
```

While the method with the bug contains parts of the evidence for the bug, it is missing the fact that the field `pointcutName` is not declared as `volatile`. So how does the model for this bug pattern achieve the surprisingly high precision and recall of 95.8% and 83.41%, respectively (Experiment SS)? We find that the correct pattern of double checked locking almost never occurs in the data set. Even the ANN search for non-buggy examples yields sequences that are indeed similar, e.g., sequences that have a null check followed by a synchronized block, but that do not exactly match the lazy initialization pattern. Given the data set, the model learns that a null check, followed by a synchronized block, followed by a another null check is likely to be buggy. In practice, this reasoning seems mostly accurate, because the idiom of double checked locking is hard to get right even for experienced programmers [Dou].

7 Note that our approach extracts the token sequence from the method body, i.e., starting from line 3. The object declaration at line 1 is shown for completeness only.

> Neural bug finding picks up signals in code that differ from the information that traditional bug detectors consider. This behavior causes the learned models to sometimes "work" even when not all of the information that is required to decide whether some code contains a bug, is available; e.g., by identifying common instances of the general bug pattern while ignoring side conditions.

6.4.4 *RQ_3: Why does neural bug finding sometimes not work?*

To answer this question, we systematically inspect false positives and false negatives for each model. We present one example for each case and provide insights why the models mis-classify them.

6.4.4.1 *Spurious Warnings*

Spurious warnings, i.e., false positives, occur when a model predicts a non-existing bug.

DEFAULT CHARSET (ID 5) In RQ_2, we showed that finding this bug pattern entails learning specific API names, e.g., `FileWriter`. Another common API that raises this warning is `String.getBytes()`, which also relies on the platform default Charset. Because this API is strongly present in the training examples, the model learns that sequences that have the `getBytes` token are likely to be buggy. However, whether an occurrence of this token is erroneous depends on whether it is the relevant method call or not, and on the receiver object on which the method is called. The following is a false positive for this bug kind, where a method with the same name is declared for a user defined type.

```
1 public class UnwovenClassFile implements IUnwovenClassFile {
2   ...
3   public byte[] getBytes() {
4     return bytes; }
5   ...
```

6.4.4.2 *Missed Bugs*

The neural models inevitably have false negatives, i.e., they fail to detect some instances of the bug patterns.

NON-OVERRIDING EQUALS (ID 11) This pattern flags methods which look like `Object.equals`, but are in fact different. A method overriding `Object.equals` must have the parameter passed to it strictly of type `Object`, a requirement for proper overload resolution. Therefore, any method that looks like `boolean equals(NotObjectType foo) {...}` should be flagged buggy. The following, is an instance of a false negative for this warning kind.

```
1 boolean equals(NodeAVL n) {
2  if (n instanceof NodeAVLDisk) {
3   return this == n ||
4     (getPos() == ((NodeAVLDisk) n).getPos()); }
5  return false; }
```

The reason why the model misses this bug is that it fails to distinguish between "`boolean equals(Object`" and any other sequence "`boolean equals(NotObjectType`". We believe that this failure is not an inherent limitation of the neural model, but can rather be attributed to the scarcity of our training data. In total, we have 165 examples of this bug in our data set, and for training the model, we use 80% of the data, i.e., around 132 examples. Given this amount of data, the recall for this bug reaches 87.88% (Experiment SS).

> Neural bug finding suffers from false positives and false negatives. The main reason is that predictions are made from partially incomplete information, e.g., due to the absence of fully qualified identifier names and types. This information is usually available to traditional bug detectors. The number of learning examples also influences the learned model.

6.4.5 *RQ_4: How does the composition of the training data influence the effectiveness of a neural model?*

To answer this question, we compare the results from Experiments BS, $B_{ANN}S$, and SS. Comparing Experiments BS and $B_{ANN}S$ in Table 6.3 shows that using ANN to select non-buggy samples for training increases the precision of the trained models in most of the cases. The reason is that having similar code examples, some of which are labeled as buggy while others are labeled non-buggy, helps the model to define a more accurate border between the two classes. Recent work on selecting inputs for testing neural networks is based on a similar observation [KFY18]. At the same time, using ANN also causes a drop in recall, mainly because the model

faces a more difficult learning task. For example, using ANN to train the model for bug pattern 2 improves precision by 5.5% but degrades recall by 2.5%.

Comparing Experiments $B_{ANN}S$ and SS shows that adding more non-buggy examples to the training set decreases the recall by a value between 2% (bug pattern 2) up to a complete erasure of the recall (bug pattern 19). On the positive side, the additional data added in Experiment SS significantly improves the precision of all models. For example, the model of bug pattern 16 improves precision by 83%.

> The composition of the training data has a huge impact. Balanced training data (Experiments BS and $B_{ANN}S$) favors recall over precision, while adding more non-buggy training data (Experiment SS) favors precision.

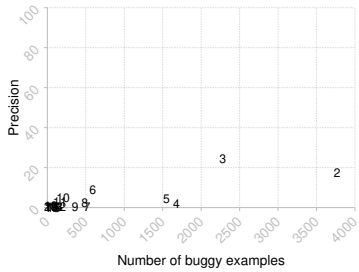
6.4.6 *RQ_5: How does the amount of training data influence the effectiveness of a neural model?*

Figure 6.2 addresses this question by plotting precision and recall of the different models over the number of buggy examples that a model is trained on in Experiments $B_{ANN}S$ and SS. All four plots show a generally increasing effectiveness, both in terms of precision and recall, for warning kinds where more data is available. For example, the models for bug patterns 2 and 3 reach high precision and recall in both experiments due to the availability of more examples. Perhaps surprisingly, though, some models are effective even with much a smaller number of warnings. For example, for bug patterns 11 and 16, the neural models achieve precision and recall above 77%, even though only 165 and 77 buggy examples are available, respectively.

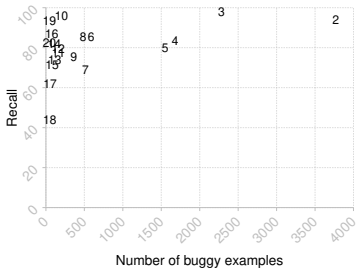
> More training data improves the effectiveness of a learned model, but surprisingly small data sets, e.g., of only 77 buggy examples, can yield reasonably effective models.

6.4.7 *RQ_6: What pitfalls exist when evaluating neural bug finding?*

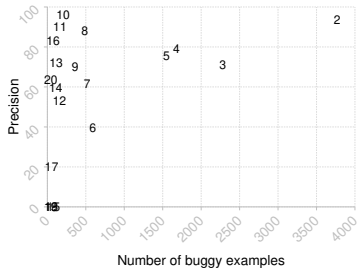
In binary classification problems, the usual setup for training and validation is to use balanced data sets. However, bugs of a specific kind are rare in real-world code. Therefore, evaluating neural bug finding and any bug



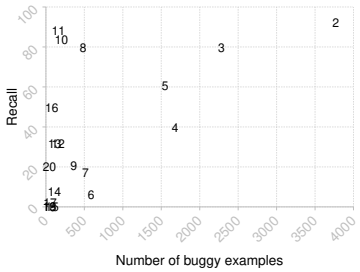
(a) Experiment $B_{ANN}S$ precision



(b) Experiment $B_{ANN}S$ recall



(c) Experiment SS precision



(d) Experiment SS recall

FIGURE 6.2: Effect of number of buggy examples on precision and recall for each warning kind. The plots use the ids from Table 6.3. Bug Id 1 is not shown due to the huge difference in x-axis scale.

finding technique using a balanced data setup yields misleading results, as described in the following.

Table 6.3 shows the results of Experiment BB, which uses balanced data for both training and validation. The first glimpse at the results is very encouraging, as they show that neural bug finding works pretty well. Unfortunately, these numbers are misleading. The reason for the spuriously good results is that the neural models overfit to the presence, or absence, of particular tokens, which may not necessarily be strong indicators of a bug.

As an example, consider bug pattern 6, which flags classes that override the `Object.equals` method but that fail to also override `Object.hashCode`. In Table 6.3, Experiment BB, the neural model predicting this warning is almost perfect with 100% recall and 98.38% precision. However, a closer look into this model and manual inspection of the training and validation examples reveal that the neural model has simply learned to predict that

the sequence of tokens "`public boolean equals (Object ...`" is always buggy. This explains why the model achieves a recall of 100%. But why is precision also quite high at 98%? It turns out that randomly sampling 590 non-buggy examples (corresponding to the number of buggy examples) from 673,446 non-buggy methods is likely to yield mostly methods that do not contain the sequence "`public boolean equal ...`". In other words, the unrealistic setup of training and validation data misleads the model into an over-simplified task, and hence the spuriously good results.

Comparing the results from Experiments BS and BB further reveals the fragility of Experiment BB's setup. In Experiment BS, the training set is constructed as in Experiment BB, but the validation set contains a lot more samples, most of them are actually not buggy. Because the models learned in Experiment BB do not learn to handle non-buggy examples similar to the buggy examples, their precision is low. That is why for the same warning kind, e.g. bug pattern 6, the precision in Experiment BS is only 8% instead of the 98.38% in Experiment BB.

> Even though bug detection can be seen as binary classification tasks, evaluating its effectiveness with balanced validation data can be highly misleading.

6.5 THREATS TO VALIDITY

Our training and validation subjects might bias the results towards these specific projects, and the findings may not generalize beyond them. We try to mitigate this problem by using the Qualitas corpus, which consists of a diverse set of 112 real-world projects.

We use warnings reported by a static analyzer as a proxy for bugs. The fact that some of these warnings may be false positives and that some actual bugs may be missed, creates some degree of noise in our ground truth. By building upon an industrially used static analyzer tuned to have less than 10% false positives [Sad+15], we try to keep this noise within reasonable bounds. Future research on collecting and generating buggy and non-buggy code examples will further mitigate this problem.

Finally, the qualitative analysis of the validation results is subject to human error. To mitigate this, two of the authors discussed and validated all the findings.

6.6 IMPLICATIONS FOR THIS DISSERTATION AND FUTURE WORK

Comparing neural bug finding head-to-head with traditional bug finders reveals that learned bug detectors show potential but still have a long way to go to be on par with their traditional counter-parts. We see empirically that neural models can learn syntactic code patterns, and hence these models are indeed capable of finding local bugs that do not require inter-procedural or type-based reasoning. At the same time, neural bug detectors often fail to detect a bug or report spurious warnings.

Our results emphasize another long-standing challenge in machine learning: Data is important. Our results demonstrate that both the amount of training data as well as how to sample the training data has a huge influence on the effectiveness of the learned bug finding models. Collecting data for neural bug finding remains an open problem, which seems worthwhile addressing in future work.

To make neural bug finding applicable to wider range of bugs, our work reveals the need for richer ML models that utilize information beyond the source code tokens, e.g., type hierarchy, data- and control-flow, and inter-procedural analysis. How to effectively feed such information into neural models is closely related to the ongoing challenge of finding suitable source code representations for machine learning. Future work should investigate how general purpose neural bug finding could benefit from such richer and more complex models.

Finally, combining API-specific knowledge, e.g., informal specification in NL or documentation, is also a natural extension to our work presented here. As shown in Chapter 4, crosschecking documentation against the runtime behavior of an API yields an effective and novel bug detection techniques. Similarly, future work based on this dissertation should examine how to augment neural static bug finding models with information extracted from documentation and other NL resources, beyond the source code itself.

6.7 CONTRIBUTIONS AND CONCLUSIONS

This chapter explores the opportunities and challenges of creating bug detectors via deep learning, a potential complement and may be an alternative to traditional bug detectors. We systematically study the effectiveness of neural bug finding based on warnings obtained from a traditional static bug detection tool. The neural bug finding approach proposed in this chapter presents a futuristic outlook for an alternative way to building static bug

finders: By learning a bug detector end-to-end from data. This idea builds on our thesis statement that learning from programs is an effective way to tackle several software engineering problems. Although the approach suffers from shortcomings, it represents early efforts addressing an open and challenging problem with a lot of untapped future work (Section 6.6 and Section 8.2).

In summary, studying neural bug detection models for 20 common kinds of programming errors shows that

- Learned bug detectors identify instances of some bug patterns with a precision of up to 73% and a recall of 97%. At the same time, the learned models struggle to find bugs of other bug patterns, which traditional analyses find easily.
- Neural bug finding works when the models learn to identify common syntactic patterns correlated with bugs, particular API misuses, or common instances of a more general bug pattern. That is, the learned models use signals in the code which are different from the information that traditional bug detectors consider.
- The size and composition of the training data a neural bug finding model is learned from have a huge impact on the model's effectiveness. More training data yields more effective models, but obtaining enough examples to train an effective neural model for finding specific bug kinds is difficult. Moreover, it is important to select buggy and correct examples that resemble each other except for the presence of a bug.
- Following a naive approach for validating a learned bug detector on balanced data may lead to very misleading results.

7

RELATED WORK

In this chapter, we discuss research work closely related to that presented in this dissertation. The literature described here is not intended to be comprehensive, rather, it highlights important related work and positions the dissertation against existing work. We begin by discussing work which focuses on various forms of software documentation, and their use in software engineering tasks. Then, we present existing work that focuses on API documentation and its use in practice. After that, we discuss work at the intersection of applying machine learning to program analysis tasks and software engineering tasks. Then, we discuss several dimensions of existing work on traditional bug detection including static and dynamic analyses, as well as anomaly detection and specification mining. Finally, we give an account of work related to static subtype checking and the usage of JSON Schema.

7.1 EXPLOITING NATURAL LANGUAGE IN SOFTWARE ENGINEERING

The central thesis of this dissertation is that leveraging software documentation provides means to equip traditional bug detection techniques with additional capabilities. Researchers and practitioners have long tried to utilize different sources of natural language to aid developers in several of their software engineering tasks, such as mining specifications from NL, finding inconsistencies between documentation and source code, and automatically generating NL descriptions and comments from programs source code.

7.1.1 *Mining Specifications from Natural Language*

Several pieces of work mine specifications or rules from natural language in API documentation, source code comments, or requirements documents. Pandita et al. [Pan+12] propose a combination of natural language processing (NLP) and heuristics to infer pre- and post-conditions from API documentation. Similarly, aComment [TZP11] infers interrupt-related pre- and post-conditions for operating systems code from NL comments and code. Doc2Spec [Zho+09a] infers resource specifications from the NL description of an API using NLP. Text2Policy [Xia+12] infers access control policies from NL documents that describe functional requirements, a combination of NLP and heuristic pattern-matching. C2S [Zha+20] translates API documentation to formal specifications in the Java modeling language (JML). The approach first builds a parallel corpus of API documentation and JML specifications of the JDK methods. Then, it uses the associated rules to synthesize specifications for new methods based on their documentation. The work presented in this dissertation is orthogonal to the above. Instead of mining natural language, we seek to enhance the documentation by learning concurrency specification based on source code properties (Chapter 3), cross-check documentation vs. runtime behavior (Chapter 4), and utilize documentation for subtype checking (Chapter 5).

Toradocu [Gof+16] and its follow-up JDoctor [Bla+18] extract executable test oracles from documentation. Toradocu focuses on whether an exception gets thrown, whereas our approach in Chapter 4 reasons about any behavior encoded in the pre- and post-state of a called method. JDoctor alleviates this limitation and infers pre- and post-conditions. The main differences to our work are: (i) Instead of aiming for general pre- and post-conditions, our work addresses an arguably simpler problem, namely predicting for a specific execution of a method whether the behavior conforms to the documentation, (ii) Instead of hard-coding heuristics, our approach learns a model from data, allowing it to generalize to patterns not supported by a finite set of heuristics, and (iii) Instead of locally reasoning about individual words or sentences and what condition they correspond to, our proposed technique globally reasons about all parts of the documentation and the full pre- and post-state of a call together. Swami [MB19] derives both test inputs and test assertions from structured NL specification of a program. Their approach relies on four patterns of NL specifications and shows that they work well for a complex subject application; yet it remains unclear whether the approach generalizes to other software.

Phan et al. [Pha+17] propose a learning-based approach to translate source code to documentation of exceptional behavior and vice versa using statistical machine translation (SMT). It is yet unknown if such technique can generate specific documentation specification, e.g., related to the concurrency behavior of a class like our work in Chapter 3.

7.1.2 *Inconsistencies Between Documentation and Code*

Other techniques aim at detecting discrepancies or inconsistencies between source code and its documentation, e.g., to detect outdated or wrong documentation. iComment [Tan+07] finds inconsistencies between comments and code through a template-based inference of programming rules. tComment [Tan+12] identifies inconsistencies between method-level comments and a method body. That approach focuses on null values and exceptions they may cause. DocRef [ZS13] combines NLP with island parsing to detect hundreds of API documentation and code inconsistencies. On a similar line of work, Ratol and Robillard [RR17] pinpoint comments that risk becoming inconsistent when changing identifier names in the code. CPC [Zha+20] propagates comments to other program elements, e.g., from a method to its callers. Using the newly propagated comments, the approach detects inconsistencies between the new comments and their corresponding source code entities. Combining that approach with our work in Chapter 4 could provide not only a much larger dataset to learn from, but also more comments to check against runtime behavior. Zhou et al. [Zho+17] generate first order logic formulas (FOL) from an API source code and another set of FOLs from the documentation by utilizing NLP techniques, such as part of speech (POS) tagging. After that, they feed the two sets of formulas into an SMT solver to detect four different kinds of inconsistencies in the API parameters documentation.

The common theme among the above approaches is that they are purely static, i.e., they analyze the source code of an API and compare it against the API documentation through pattern matching and NLP techniques. Our approach in Chapter 3 is similar in that it also extracts features from source code statically. However, it differs in various aspects: (i) It tackles an understudied problem in documentation: thread safety of object oriented classes, (ii) It does not compare code against documentation, rather, it learns from source code only, and (iii) It relies on automatic learning, without using any hand-crafted rules or heuristics. Moreover, our approach for crosschecking documentation versus runtime presented in Chapter 4

distinguishes itself from the above by utilizing the runtime behavior of the API instead of its source code, and it also utilizes learning instead of template-based patter matching.

7.1.3 *Learning from Natural Language*

Recently, researchers started to tackle several software engineering problems by applying machine learning to programs. Several of such approaches utilize NL in source code, e.g., documentation to infer type signatures of JavaScrip [MPP19] and Python functions [Pra+20], or utilize the source code itself to recover meaningful and useful NL information, e.g., JavaScript identifier names in obfuscated JavaScript code [VCD17]. We give a more detailed account of various applications of machine learning in program analysis in Section 7.3.

7.2 API DOCUMENTATION IN PRACTICE

API documentation has received a lot of attention because it is pervasive, often rich in information, and represents the entry point for new developers when they learn about a new API.

7.2.1 *Studies of API Documentation*

Monperrus et al. [Mon+12] analyzed hundreds of Java API documentation items and identified 23 different directives which occur throughout the analyzed documentation. Lethbridge, Singer, and Forward [LSF03] study how documentation is used in practice. They find that documentation is often outdated and inconsistent. Similarly, Aghajani et al. [Agh+19] analyzed hundreds of software artifacts related to documentation such as Stack Overflow questions, pull requests, issues, and mailing lists; and built a taxonomy of documentation issues. They identify several issues related to the correctness, completeness, staleness, and usability of the documentation. In their follow-up work, Aghajani et al. [Agh+20] surveyed developers and practitioners regarding the documentation issues they identified earlier and confirmed that developers are very interested in tool support for automatic generation and maintenance of documentation. Another study focuses on problems that developers face when learning a new API [Rob09]. Their results include that many APIs need more and better documentation. Such studies help guide future work on API documentation. Our contributions in

Chapter 3 and Chapter 4 for inferring concurrency specification from source code and cross-checking documentation vs. runtime behavior, respectively, provide automated techniques to alleviate some of issues raised by those studies.

7.2.2 *Enhancing the Usage of API Documentation*

Improving documentation and how developers use it is one of the active areas of research. McBurney et al. [McB+17] investigate how to prioritize documentation effort based on source code attributes and textual analysis. Treude and Robillard [TR16] augment API documentation with relevant and otherwise missing information from Stack Overflow. APIBot is a bot created to answer NL questions by developers based on the available documentation [Tia+17]. Other work finds relevant tutorial fragments for an API to help developers better understand that API [Jia+17]. Our work in Chapter 3 contributes to improving and adding otherwise missing documentation, yet we tackle the so far understudied problem of inferring concurrency-related documentation. Additionally, in Chapter 4 we propose an approach to automatically reason about documentation w.r.t. the observed program runtime behavior.

7.3 MACHINE LEARNING AND PROGRAM ANALYSIS

The main thesis of this dissertation is that learning from programs and their documentation provide means to improving traditional bug finding and prevention techniques. Our learning based approaches presented in Chapters 3, 4, and 6 relate to a recent stream of machine learning-based program analyses. We give an account on some of the work related to ours while we refer the reader to the work of Allamanis et al. [All+18] which provide a more detailed view of the different lines of work related to the idea of applying ML to program analysis.

7.3.1 *Program Representation for Learning*

A challenge for any ML-based program analysis is how to represent a program for the learning task. Work on this problem includes graph-based representations [ABK17; Bro+18], embeddings learned from sequences of API calls [DTR18], embeddings learned from paths through ASTs [Alo+18a; Alo+18b], and embeddings for edits of code [Yin+18]. A hybrid represen-

tation comprised of symbolic execution and concrete execution traces has been proposed recently [WS20] with its authors arguing that it is more effective than, for instance, representations learned from source code only. In Chapter 4, we represent the runtime state of the program execution by a token-based sequence of the key-value pairs of the program serialized object. In Chapter 6, we opt for a simple representation for programs source code, the sequence of their source code tokens. Future work could possibly explore whether our learning based approaches proposed in this dissertation could benefit from other, possibly richer, representations of programs.

GRAPH KERNELS Our technique in Chapter 3 uses a novel graph representation for the program source code, the field-focused graph, and a classical graph embeddings, the Weisfeiler-Lehman graph kernels [She+11]. Kondor and Lafferty [KL02] and Gärtner, Flach, and Wrobel [GFW03] introduced the concept of graph kernels, and various other kinds of graph kernels have been proposed since then, e.g., random-walk kernels [KTI03], shortest-path kernels [BK05], and subtree kernels [RG03]. These graph kernels have been mainly used in bioinformatics [Bor+05], in chemoinformatics [Ral+05; Swa+05], and in web mining [WM03], e.g., to find similar web pages and to analyze social networks.

Some existing work utilizes graph kernels for software engineering tasks. Wagner et al. [Wag+09] analyze process trees with graph kernels to identify malware. Another approach [And+11] uses Markov chains constructed from instruction traces of executables [And+11]. Furthermore, graph kernels have been applied to statically identify malware by applying a neighborhood hash graph kernel on call graphs [Gas+13] and by using graph edit distance on API dependency graphs [Zha+14]. Our TSFinder proposed in Chapter 3 tackles a different problem: the lack of documentation regarding multi-threaded behavior. Another difference is the kind of information that TSFinder extracts from classes and then feeds into graph kernels. Finally, to the best of our knowledge, our experimental setup is orders of magnitude larger than any other study that utilizes graph kernels in the context of program analysis.

7.3.2 *Learning to Find Bugs*

Learned models are becoming increasingly popular for bug finding, motivated by the advances in ML and the observation by Ray et al. [Ray+16] that

buggy code is less frequent than non-buggy code. DeepBugs exploits identifier names, e.g., of variables and methods, to find buggy code [PS18]. Vasic et al. [Vas+19] use pointer networks to jointly find and fix variable mis-use bugs. Weston, Chopra, and Bordes [WCB14] train a memory network to predict whether a piece of code may cause a buffer overrun [Cho+17]. Wang et al. [Wan+19] use graph neural networks (GNN) [Sca+09] to detect three kinds of bug patterns. A broader set of coding mistakes that may cause vulnerabilities is considered in other learning-based work [Li+18].

These approaches focus on neural models that detect specific kinds of bug, in the order of two or three kinds at most. Our work in Chapter 6 contributes a broader study of neural bug finding and a head-to-head comparison with a state-of-the-art bug detector. A study related to ours applies different learning techniques to the bug detection problem [Cha+17]. The authors of that study use a data set that includes seeded bugs, whereas we use real bugs. Another difference is that most of their study uses manually extracted features of code, whereas we learn models fully automatically, without any feature engineering. Their preliminary results with neural networks are based on a bit-wise representation of source code, which they find to be much less effective than we show sequence of tokens-based models to be.

7.3.3 *Learning from Source Code*

To exploit the observation of Hindle et al. [Hin+12] that source code is repetitive, even more than natural language, researchers and practitioners apply machine learning to source code for a variety of tasks. Some of the machine learning-based program analyses, beyond bug detection, include predicting types [Hel+18; MPP19; Pra+20; RVK15], improving or predicting identifier names [All+15; Liu+19; RVK15; VCD17], detecting clones [Whi+16; ZH18], searching code [GZK18; Sac+18], predicting code edits [Tuf+19; Yin+18; Zha+18], reasoning about code edits [Hoa+20], classifying code [Mou+16; Zha+19], automatically fixing bugs [Bad+19; Gup+17; Har+18], and generating unit tests assertion statements [Wat+20]. Those approaches, similar to Chapters 3 and 6, are purely static, in contrast to our dynamic approach in Chapter 4.

7.3.4 *Learning from Program Execution*

Beside learning from source code, researchers also examine learning from different aspects of program execution. Vanmali, Last, and Kandel [VLK02]

train a model that, given the input, predicts the output of a program. Such a model can serve as a test oracle for one specific program, but in contrast to our work in Chapter 4 does not generalize across programs. Piech et al. [Pie+15] learn program embeddings based on input-output relations gathered during executions, and use the embeddings to provide feedback on student code. Wang, Singh, and Su [WSS17] learn an embedding of programs from dynamically gathered sequences of program states, and use the embedding to classify programs based on the mistakes they contain. Tsimpourlas, Rajan, and Allamanis [TRA20] describe a neural model that classifies traces of test executions as failing or passing. In contrast to DocRT, their approach does not leverage NL information and uses traces with all calls that happen within the software under test. All the above work shares the idea to learn from program executions, but uses a different approach and has a different purpose than our crosschecking documentation versus runtime in Chapter 4. The direction of learning from program executions is a new one and several ideas, such as the kind of runtime information and how to represent it, are yet to be explored further.

7.4 TRADITIONAL BUG DETECTION TECHNIQUES

Software developers have long relied on several techniques to help them detect and prevent bugs in programs before they put a program in production or ship it to end users. Among the different techniques, we focus here on the two most relevant families of bug detection mechanisms: static analysis and dynamic analysis. In Chapter 2, we analyze the effectiveness of state-of-the-art static bug detectors. Moreover, the approaches we propose in Chapters 3, 5, and 6 are static while the approach presented in Chapter 4 is dynamic.

7.4.1 *Static Analysis*

The lint tool [Joh78], originally presented in 1978, is one of the pioneers on static bug detection. Since then, static bug detection has received significant attention by researchers and industry, including work on finding API misuses [Ngu+09; Pra+12; WZ09], name-based bug detection [PG11], security bugs [Bro+17], and on detecting performance bugs [PH13]. These approaches, similar to our work in Chapter 6, target various kinds of software bugs. Ours, however, is different in that it does not rely on any static analysis, but just utilizes warnings produced by static checkers to collect

data to train a machine learning model. Yet, our technique does not come on par with such mature tools yet in terms of its precision and recall, which calls for more research in this direction. Furthermore, our work in Chapter 5 takes a different approach to static bug detection by introducing and utilizing a static subtype checker which finds data incompatibility bugs, a class of bugs usually not detected by the above static bug finders.

Additionally, there are various static analyses of concurrent code, e.g., to find deadlocks [AWS05; Nai+09; WTE05], atomicity violations [FQ03], locking policies [FF00], and conflicting objects [PG03]. One strength of our approach in Chapter 3 compared to existing static analyses of concurrent code is the use of a relatively simple static analysis and complementing it with graph-based machine learning.

REAL-WORLD DEPLOYMENTS OF STATIC ANALYSES Several static bug detection approaches have been adopted by major industry players. Bessey et al. [Bes+10] report their experiences from commercializing static bug detectors. Ayewah and Pugh [AP10] and Ayewah et al. [Aye+08] share lessons learned from applying FindBugs, the predecessor of the SpotBugs tool considered in our study in Chapter 2, at Google. More recent tools deployed in industry include Error Prone [Aft+12], which is used at Google and serves as an oracle for our work in Chapter 6, and Infer [Cal+15], which is used at Facebook. Rice et al. [Ric+17] describe the success of deploying a name-based static checker at Google too. Those industrial bug finders, namely Error Prone, Infer, and SpotBugs, are evaluated in our study in Chapter 2. Moreover, those rule-based approaches involve significant manual effort for creating and tuning the bug detectors, whereas in Chapter 6, our bug detectors are learned from examples only.

PRIORITIZING STATIC ANALYSIS WARNINGS Since many bug detectors suffer from a large number of warnings, some of which are false positives, an important question is which warnings to inspect first. Work on prioritizing analysis warnings addresses this question based on the frequency of true and false positives [KE03], the version history of a program [KE07], and statistical models based on features of warnings and code [Rut+08]. These efforts are orthogonal to the bug detection problem addressed in this dissertation, and could possibly be combined with several of the approaches presented here.

7.4.2 *Dynamic Analysis*

Automated test generation has received wide attention, and there are various techniques for unit-level testing, e.g., random-based [Ciu+08; CS04; Pac+07], search-based [FA11], or based on symbolic [CDE08; Kin76] and concolic [GKS05; SMA05] execution [CS13; Thu+11; Xie+05]. Automated testing has also been applied, e.g., to concurrent software [CLP17; PG12] and to graphical user interfaces [CNS13; CGO15; GFZ12; Pra+14]. Most of the existing test generators focus on generating test inputs that cover as much behavior of the software under test as possible. Our work in Chapter 4 provides a novel way for detecting bugs by learning to crosscheck an API documentation against its observed runtime behavior and hence, it provides an orthogonal contribution that could be combined with these test generators.

DYNAMIC ANALYSIS FOR CONCURRENCY The analysis of concurrent software has been an active topic for several years. Analyses that target thread-safe classes are particularly related to our work. ConTeGe [PG12] and Ballerina [Nis+12] have pioneered test generation for such classes. Other test generators improve upon them by considering coverage information [CLP17; TC16], by steering test generation based on sequential test executions [SR14; SR15; SRJ15], by comparing thread-safe classes against their superclasses [PG13], or by targeting tests that raise exceptions [STR16]. SpeedGun [PHG14] detects performance regression bugs in thread-safe classes. ConCrash [BPT17] creates tests that reproduce previously observed crashes. LockPeeker [Lin+16] tests API methods to find latent locking bugs.

Beyond thread-safe classes, various dynamic analyses to find concurrency bugs have been proposed, such as data race detectors [Cho+02; FF09; OC03; Sav+97], analyses to detect atomicity violations [AHB03; FF04; Lu+06; WS06; XBH05], and analyses to find other kinds of concurrency anomalies [LC09; PG01]. While these techniques analyze a given execution, another direction is to influence the schedule of an execution to increase the chance to trigger concurrency-related misbehavior. Work on influencing schedules includes random-based scheduling [Bur+10; Ede+02], systematic exploration of schedules [Mus+08; Vis+03], and forcing schedules to trigger previously identified, potential bugs [Jos+09; Sen08]. All these approaches find correctness or performance bugs in thread-safe classes or general concurrency-related bugs. Instead, our work in Chapter 3 addresses the

orthogonal problem of inferring whether a class is even supposed to be thread-safe.

THE TEST ORACLE PROBLEM Various approaches to address the test oracle have been proposed, as surveyed by Barr et al. [Bar+15]. Commonly used approaches include regression testing [LW90; Rot+01], differential testing [McK98], specification mining [ABL02; Dal+06; LZ05; PG09; Sho+07; Yan+06], and metamorphic testing [CCY20; Seg+16]. Our work in Chapter 6 differs from these lines of work by exploiting NL information and by learning an oracle from a large number of executions.

7.4.3 *Studies of Bug Detection Techniques*

One of the contributions of this dissertations is the study of static bug finders and their effectiveness, presented in Chapter 2. In this section we examine the most relevant studies which also look at how static and dynamic bug detection techniques perform in practice.

STUDIES OF STATIC BUG DETECTION Most existing studies of static bug detectors focus on precision, i.e., how many of all warnings reported by a tool point to actual bugs [RAF04; Wag+05; Zhe+06]. To address that question, the tools are applied to code bases and then all warnings, possibly after some automated filtering, are manually inspected. In contrast, our study in Chapter 2 asks the opposite question: What is the recall of static bug detectors, i.e., how many of all (known) bugs are found? The reason why most existing studies neglect this question is that answering it requires knowing bugs that have been found independently of the studied bug detectors. Another difference to existing studies is our choice of static bug detectors: To the best of our knowledge, this is the first study to evaluate the effectiveness of Error Prone, Infer, and SpotBugs.

The most related existing work is a study by Thung et al. [Thu+12; Thu+15] that also focuses on the recall of static bug detectors. Our work differs in the methodology used to answer this question: We manually validate whether the warnings reported by a tool correspond to a specific bug in the code, instead of checking whether the lines flagged by a tool include the faulty lines. This manual validation leads to significantly different results than the previous study because many warnings coincidentally match a faulty line but are actually unrelated to the specific bug. While they conclude that between 64% and 99% of all bugs are partially or fully

detected, we find that only 4.5% of all studied bugs are found. The main reason for this difference is that some of the bug detectors used by Thung et al. report a large number of warnings. For example, a single tool alone reports over 39,000 warnings for the Lucene benchmark (265,821 LoC), causing many lines to be flagged with at least one warning with error rate 0.15. Since their methodology fully automatically matches source code lines and lines with warnings, most bugs appear to be found. Instead, we manually check whether a warning indeed corresponds to a particular bug to remove false matches. Another difference is that our study focuses on a more recent, improved, and industrially used static bug detectors. Thung et al.'s study considers what might be called the first generation of static bug detectors for Java, e.g., PMD and CheckStyle. While these tools contributed significantly to the state-of-the-art when they were initially presented, it has also been shown that they suffer from severe limitations, in particular, large numbers of false positives. Huge advances in static bug detection have been made since then. Our study focuses on a novel and improved generation of static bug detectors, including tools that have been adopted by major industry players and that are in wide use.

Rahman et al. [Rah+14] compare the benefits of static bug detectors and statistical bug prediction. To evaluate whether an approach would have detected a particular bug, their study compares the lines flagged with warnings and the lines changed to fix a bug, which roughly corresponds to the first step of our methodology and lacks a manual validation whether a warning indeed points to the bug. Johnson et al. [Joh+13] conducted interviews with developers to understand why static bug detectors are (not) used. The study suggests that better ways of presenting warnings to developers and integrating bug detectors into the development workflow would increase the usage of these tools.

STUDIES OF DYNAMIC BUG DETECTION The effectiveness of test generation techniques has been studied as well [Alm+17; Sha+15]. One of these studies [Sha+15] also considers bugs in Defects4J and finds that most test generators detect less than 20% of these bugs. Our work in Chapter 4 complements those studies by systematically studying neural bug finding. Finally, Legunsen et al. [Leg+16] study to what extent checking API specifications via runtime monitoring reveals bugs. All these studies are complementary to ours, as we focus on static bug detectors. Future work could study how different bug finding techniques complement each other.

STUDIES AND DATASETS OF BUGS AND BUG FIXES An important step toward improving bug detection is to understand real-world bugs. To this end, studies have considered several kinds of bugs, including bugs in the Linux kernel [Cho+01], concurrency bugs [Lu+08], and correctness and performance bugs [Oca+13; SP16] in JavaScript. Pan, Kim, and Jr [PKJ09] study bug fixes and identify recurring, syntactical patterns. BugBench [Lu+05] consists of 17 bugs in C programs. Cifuentes et al. [Cif+09] significantly extend this benchmark, resulting in 181 bugs that are sampled from four categories, e.g., buffer overflows. They use the benchmark to compare four bug detectors using an automatic, line-based matching to measure recall. Future work could apply our semi-manual methodology in Chapter 2 to their bug collection to study whether our results generalize to C programs.

Defects4J [JJE14], the dataset we use in our study in Chapter 2, consists of several hundreds real-world Java bugs along with their real fixes by the developers. These bugs have been manually curated and their fixes have been minimized to isolate the bug fix from any non-relevant code change. More recently, Tomassi et al. [Tom+19] introduced the BugSwarm dataset. BugSwarm provides orders of magnitude more bugs than Defects4J [JJE14] but was developed and released after we concluded our study and it was not available to us at that time. Future work could extend our study by using the BugSwarm dataset and studying the generalization of our results.

7.4.4 *Defect Prediction*

Orthogonal to bug detection is the problem of defect prediction [FN99; Zim+09]. Instead of pinpointing specific kinds of errors, as our work, it predicts whether a given software component will suffer from any bug at all. Li et al. [Li+19] and Wang, Liu, and Tan [WLT16] propose neural network-based models for this task. Harer et al. [Har+18] train a convolution neural network (CNN) to classify methods as vulnerable or not based on heuristics built on labels from a static analyzer, similar to what we do in Chapter 6. These approaches and some of our work (Chapters 4 and 6) share the idea of formulating bug detection as a classification problem. However, our neural bug finding approach in Chapter 6 tackles a much more challenging problem: Classifying a specific piece of code as buggy or not w.r.t to a specific bug kind, i.e., it tries to predict the bug kind instead of predicting just buggy or not. Moreover, our work in Chapter 4 does not analyze source code, instead, our model learns to cross-validate runtime behavior against documentation.

UNBALANCED DATA Machine learning models for software defect prediction [Son+11] suffer from data imbalance [WY13] as we also note in Chapter 6. Skewed training data is usually tackled by different approaches such as sampling techniques [Kam+07], cost-sensitive learning [Sun+07], or ensemble learning [SSZ12]. Under-, over-, or synthetic-sampling [Ben+18; Kam+07] have been applied to alleviate data imbalance in software defect prediction. Our approach in Chapter 6 leverages approximated nearest neighbor (ANN) sampling of non-buggy examples, a form of guided undersampling.

7.5 ANOMALY DETECTION AND SPECIFICATION MINING

Another line of work related to that presented in this dissertation focuses on mining patterns from source code, runtime, and other software artifacts. The mining process is often termed anomaly detection when its purpose is to detect outliers or anti-patterns. When the mining aims at extracting frequent patterns, practitioners call it specification mining.

7.5.1 *Specification Mining*

Specification mining automatically extracts a formal specification from source code or from programs executions. Mined specification include temporal specifications of API usages [GS08; Ngu+09; WZ09; Yan+06; ZZM08; Zho+09b], finite-state specifications of method calls [ABL02; LCR11; PG09; WML02], algebraic specifications [HRD08], locking disciplines [Ern+16], exception-handling rules [TX09], semantic code-change patterns [Ngu+19], and implicit programming rules [LZ05]. One benefit of mined specifications is to use them as documentation. Our TSFinder presented in Chapter 3 can be seen as a form of specification mining. In contrast to existing techniques, our work focuses on concurrency documentation and uses machine learning to learn from known examples how to infer this specification (documentation).

7.5.2 *Anomaly Detection*

Anomaly detection approaches search for code or runtime behavior that stands out and therefore, may be buggy or malicious. Monperrus, Bruch, and Mezini [MBM10] learn objects usage patterns to detect likely missing method calls in specific code locations. Bugram uses a statistical lan-

guage model that warns about uncommon n-grams of source code tokens [Wan+16]. Salento learns a probabilistic model of API usages and warns about unusual usages [MCJ17]. Gruska, Wasylkowski, and Zeller [GWZ10] and Wasylkowski, Zeller, and Lindig [WZL07] mine objects and interfaces usage models, respectively, and label outliers as potential usage anomalies. Zhang et al. [Zha+14] mine contextual API dependency graphs to detect Android malware. Ray et al. [Ray+16] explain why this is possible and show that buggy code is less natural than non-buggy code.

To detect anomalies at runtime, several approaches track and mine different runtime artifacts, e.g., heap properties [CG06] and properties of objects and variables at instrumented code locations [HL02], to detect buggy software. Moreover, other mining techniques are used to detect abnormal or malicious behavior, e.g., based on finite-state automata of system calls [Sek+01], abstract execution paths from stack traces [Fen+03], and execution graph of system calls [GRS04]. Our work in Chapter 4 could be viewed as a kind of runtime anomaly detection where execution behavior combined with documentation are used as features to detect unlikely behavior or wrong documentation. However, ours differs from the above mentioned approaches in that it leverages NL documentation and similar to our work in Chapter 6, it learns from buggy and non-buggy examples.

7.6 JSON SCHEMA AND SUBTYPE CHECKING

In this section, we discuss work related to the JSON Schema subtype checking presented in Chapter 5.

7.6.1 *JSON Schema Subtyping and Formalism*

Practitioners have significant interest in reasoning about the subtype relation of JSON schemas. In Chapter 5, we present an experimental comparison against isSubset [Hag19], which is the closest tool to our work and was developed concurrently with ours. Another closely related tool [Jsob] relies on simple syntactic checks. That work considers a change as a breaking change whenever a node is removed from the schema. As illustrated in Section 5.2.3, removing nodes (or replacing them by others) may yield not only subtypes but even equivalent schemas. Yet another existing tool [Jsoa] checks whether two schemas are equivalent but does not address the subtyping problem. We are not aware of any research efforts done in the

direction of defining and checking the subtype relation on JSON schemas (subschema).

Pezoa et al. [Pez+16] formally define the syntax and semantics of JSON Schema, including the JSON validation problem. An alternative formulation of JSON validation uses a logical formalism [Bou+17]. Baazizi et al. [Baa+17] address the problem of inferring schemas for irregular JSON data, but their work does not use the JSON Schema standard we are targeting here. None of the above pieces of work addresses the subschema problem.

There are other schema definition languages for JSON besides JSON Schema, e.g., Avro [Apa]. Protobuf [Pro] is Google's data exchange format, which borrows several features from JSON schema. Our work might help define subtype relations for these alternative languages.

The Swagger (OpenAPI) specification [Swab] uses JSON Schema to define the structure of RESTful APIs. The swagger-diff tool [Swaa] aims at finding breaking API changes through a set of syntactic checks, but does not provide the detailed checks that we do. Our work in Chapter 5 could be integrated as part of the pipeline to check for subtle backward compatibility breaking-changes.

7.6.2 *Applications of Subschema Checks*

One application of JSON subschema is for statically reasoning about breaking changes of web APIs. A study of the evolution of such APIs shows that breaking changes are frequent [Li+13]. Another study reports that breaking changes of web APIs cause distress among developers [EZG15]. Since JSON schemas and related specifications are widely used to specify web APIs, our approach can identify breaking changes statically instead of relying on testing.

Data validation for industry-deployed machine learning pipelines is crucial as such pipelines are usually retrained regularly with new data. To validate incoming data, Google TFX [Bay+17] synthesizes a custom data schema based on statistics from available data and uses this schema to validate future data instances fed to the TensorFlow pipeline [Bre+19]. Amazon production ML pipelines [Sch+18] offer a declarative API that lets users manually define desired constraints or properties of data. Then data quality metrics, such as completeness and consistency, are measured on real-time data with respect to the pre-defined constraints, and anomalies are reported. Both systems are missing an explicit notion of schema subtyping. For instance, TFX uses versioned schemas to track the evolution of inferred data

schemas, and reports back to the user whether to update to a more (or less) permissive schema based on the historical and new data instances [Bay+17]. Lale uses JSON schemas to specify both correct ML pipelines and their search space of hyperparameters [Hir+19]. The ML Bazaar also specifies ML primitives via JSON [Smi+19]. Another type-based system for building ML pipelines is described in [PKN16]. This kind of system can benefit from JSON subschema checking to avoid running and deploying incompatible ML pipelines.

7.6.3 *Type Systems for XML, JavaScript, and Python*

Semantic subtyping [CF05] handles Boolean connectives on types by using a disjunctive normal form similar to that in Chapter 5. It was developed in the context of CDuce, a functional language for working with XML [BCF03]. Subschema checking for XML, called schema containment, is also addressed in [TH03]. XDuce is a static language for processing XML documents using XML schemas as types [HP03] which makes use of "regular expression types" [HVP05]. Our work differs in working on JSON, not XML, which has a different feature set. Moreover, these approaches treat XML as tree automata, shown to be less expressive than JSON Schema [Pez+16].

Both JavaScript and Python have a convenient built-in syntax for JSON documents. Furthermore, there are type systems retrofitted onto both languages [BAT14; Vit+14]. Therefore, a reasonable question to ask is whether JSON schema subtype queries could be decided by expressing JSON schemas in those languages and then using the subtype checker of those type systems. Unfortunately, this is not the case, since JSON Schema contains several features that those type systems cannot express, such as negation, `multipleOf` on numbers, and `pattern` on strings.

8

CONCLUSIONS AND FUTURE WORK

In this chapter, we conclude this dissertation by summarizing the high level contributions and highlighting important future work in several directions.

8.1 SUMMARY OF CONTRIBUTIONS

This dissertation shows that learning from programs and their documentation provides an effective means to prevent and detect software bugs. Our contributions include techniques that find bugs and inconsistencies in programs and their documentation. More concretely, this dissertation contributes the following:

1. An extensive study of the recall of state-of-the-art static bug finding tools on a large set of real-world bugs. Our study reveals the low recall of modern static bug detectors and the eminent need to improve them. In particular, the current tools fail to detect several bugs that require domain-specific knowledge to be detected correctly.

2. TSFinder, an effective learning-based approach to infer concurrency specifications, i.e., thread-safety documentation, of object-oriented classes. Our approach provides otherwise missing documentation which could impact the correctness and performance of the underlying software, when used incorrectly.

3. DocRT, a learning-based approach to crosscheck the documentation of an API against its observed exceptional runtime behavior. Our approach successfully detects a large set of known bugs in addition to finding several new ones in both the documentation and implementation of APIs.

4. jsonSubSchema, an algorithm that leverages the JSON Schema descriptions of APIs for static subtype checking. This approach detects a new class of data-compatibility bugs which affects applications across several domains including the Web and machine learning libraries.

5. Neural bug finding, a futuristic approach for creating static bug detectors by learning end-to-end from examples only. This promising approach, though not on par yet with traditional bug detection techniques, shows potential and opens the door for several directions in future work, as we discuss next.

8.2 FUTURE WORK

As shown in Chapter 7, finding bugs in programs is an ever going research problem. Applying machine learning to solve some of the challenges for traditional program analyses already started to gain momentum [All+18]. Along the same line, the work presented in this dissertation reveals several potential directions for future work and lessons learned which we identify and highlight in the following.

- *Software documentation is under-utilized*. Documentation is an important source of valuable information for the software development process. We need to design new methods to automatically leverage the knowledge embedded in software documentation to improve the quality of the programs we use in our everyday life.

- *Software documentation suffers from several shortcomings*. Developers struggle with unclear, incomplete, stale, and inconsistent documentation which eventually reflects on the quality of software they develop. We need to develop more efficient and automated tools to improve software documentation.

- *Learning from programs is promising, but has a long way to go*. This is an ongoing research problem with a lot of interesting work. However, the current state-of-the-art is far from being perfect. We need richer representations of programs, whether source code or runtime behavior, to enable more effective learning and unlock more applications of machine learning to program analysis.

- *Data for learning from programs is still an open problem*. This is a challenge for any machine learning application. However, one would

think that it is not the case for software engineering tasks because of the abundance of available open-source code. In part, this is true for unsupervised learning tasks, but it is not the case for supervised ones like our work in Chapters 3, 4, and 6. Many interesting software engineering tasks would require a lot of labeled data to enable machine learning solutions.

- *A fair experimental setup for ML-based program analysis is tricky*. We have seen in Chapter 6 that the data splits for training and validation have huge impact on the accuracy, precision, and recall of learned models of source code. Therefore, researchers and practitioners should be more careful when designing their experimental setups to evaluate ML models for programs.

- *Learning from programs and their documentation is promising*. This is the central thesis of this dissertation. We argue that there is a lot of potential to be unlocked in this direction. One of the main motivations to pursue this idea is that programs are closely coupled with their documentation although they are two different communication channels: One is intended for humans and the other is more geared towards machines. But as programs are mostly written by humans, this dissertation has shown that both mediums can benefit each other.

BIBLIOGRAPHY

[Aft+12] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. "Building Useful Program Analysis Tools Using an Extensible Java Compiler". In: *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 2012, 14.

[AWS05] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. "Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring". In: *Haifa Verification Conference*. Vol. 3875. Springer, 2005, 191.

[Agh+19] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, 1199.

[Agh+20] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. "Software Documentation: The Practitioners' Perspective". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, 590–601.

[ABK17] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *CoRR* abs/1711.00740 (2017).

[All+18] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), 81.

[All+15] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. "Suggesting accurate method and class names". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 2015, 38.

[Alm+17] Mohammad Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. 2017, 263.

[Alo+18a] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "A General Path-Based Representation for Predicting Program Properties". In: *PLDI*. 2018.

[Alo+18b] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "code2vec: Learning Distributed Representations of Code". In: *CoRR* arXiv:1803.09473 (2018).

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. "Mining specifications". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2002, 4.

[And+11] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. "Graph-based malware detection using dynamic analysis". In: *Journal in Computer Virology* 7.4 (2011), 247.

[Apa] *Apache Avro*. URL: http://avro.apache.org/.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. "High-level data races". In: *Software Testing, Verification and Reliability* 13.4 (2003), 207.

[AP10] Nathaniel Ayewah and William Pugh. "The Google FindBugs fixit". In: *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*. 2010, 241.

[Aye+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008), 22.

[Baa+17] Mohamed Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. "Counting types for massive JSON datasets". In: *Symposium on Database Programming Languages (DBPL)*. 2017, 9:1.

[BJR19] Hlib Babii, Andrea Janes, and Romain Robbes. "Modeling Vocabulary for Big Code Machine Learning". In: *CoRR* (2019).

[Bad+19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. "Getafix: Learning to Fix Bugs Automatically". In: *OOPSLA*. 2019.

[Bar+15] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. Software Eng.* 41.5 (2015), 507.

[Bay+17] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform". In: *Conference on Knowledge Discovery and Data Mining (KDD)*. Halifax, NS, Canada, 2017, 1387.

[Ben+18] Kwabena Ebo Bennin, Jacky Keung, Passakorn Phannachitta, Akito Monden, and Solomon Mensah. "Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction". In: *IEEE Transactions on Software Engineering* 44.6 (2018), 534.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. "CDuce: an XML-centric general-purpose language". In: *International Conference on Functional Programming (ICFP)*. 2003, 51.

[Bes+10] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. "A few billion lines of code later: Using static analysis to find bugs in the real world". In: *Communications of the ACM* 53.2 (2010), 66.

[BS16] Sahil Bhatia and Rishabh Singh. "Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks". In: *CoRR* abs/1603.06129 (2016).

[BPT17] Francesco A. Bianchi, Mauro Pezze, and Valerio Terragni. "Reproducing Concurrency Failures from Crash Stacks". In: *FSE*. 2017.

[BAT14] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. "Understanding TypeScript". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2014, 257.

[Bla+18] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. "Translating code comments to procedure specifications". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, 242.

[Boj+17] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. "Enriching Word Vectors with Subword Information". In: *TACL* 5 (2017), 135.

[BK05] Karsten M. Borgwardt and Hans-Peter Kriegel. "Shortest-Path Kernels on Graphs". In: *Proceedings of the Fifth IEEE International Conference on Data Mining*. ICDM '05. Washington, DC, USA: IEEE Computer Society, 2005, 74.

[Bor+05] Karsten M. Borgwardt, Cheng Soon Ong, Stefan Schönauer, S. V. N. Vishwanathan, Alex J. Smola, and Hans-Peter Kriegel. "Protein Function Prediction via Graph Kernels". In: *Bioinformatics* 21.1 (2005), 47.

[Bou+17] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. "JSON: Data model, Query languages and Schema specification". In: *Symposium on Principles of Database Systems (PODS)*. 2017, 123.

[Bra+02] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. "GraphML Progress Report Structural Layer Proposal". In: *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, 501.

[Bre+19] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. "Data Validation for Machine Learning". In: *Conference on Systems and Machine Learning (SysML)*. 2019.

[Bro+18] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. "Generative Code Modeling with Graphs". In: *ArXiv e-prints* (2018).

[Bro+17] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. "Finding and Preventing Bugs in JavaScript Bindings". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 2017, 559.

[Bur+10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. "A randomized scheduler with probabilistic guarantees of finding bugs". In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010, 167.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008, 209.

[CS13] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Commun. ACM* 56.2 (2013), 82.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving fast with software verification". In: *NASA Formal Methods Symposium*. Springer. 2015, 3.

[CF05] Giuseppe Castagna and Alain Frisch. "A Gentle Introduction to Semantic Subtyping". In: *Symposium on Principles and Practice of Declarative Programming (PPDP)*. 2005, 198.

[Cha+17] Timothy Chappelly, Cristina Cifuentes, Padmanabhan Krishnan, and Shlomo Gevay. "Machine learning for finding bugs: An initial report". In: *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE. 2017, 21.

[CCY20] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. "Metamorphic testing: a new approach for generating next test cases". In: *arXiv preprint arXiv:2002.12543* (2020).

[CG06] Trishul M. Chilimbi and Vinod Ganapathy. "HeapMD: identifying heap-based bugs using anomaly detection". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, 219.

[Cho+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. "Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2002, 258.

[Cho+17] Min je Choi, Sehun Jeong, Hakjoo Oh, and Jaegul Choo. "End-to-End Prediction of Buffer Overruns from Raw Source Code via Neural Memory Networks". In: *CoRR* abs/1703.02458 (2017).

[CNS13] Wontae Choi, George Necula, and Koushik Sen. "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2013, 623.

[Cho+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. "An Empirical Study of Operating System Errors". In: *Symposium on Operating Systems Principles (SOSP)*. 2001, 73.

[CLP17] Ankit Choudhary, Shan Lu, and Michael Pradel. "Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests". In: *International Conference on Software Engineering (ICSE)*. 2017, 266.

[CGO15] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. "Automated Test Input Generation for Android: Are We There Yet? (E)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 2015, 429.

[Cif+09] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. "BegBunch: Benchmarking for C bug detection tools". In: *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM. 2009, 16.

[Ciu+08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. "ARTOO: adaptive random testing for object-oriented software". In: *International Conference on Software Engineering (ICSE)*. ACM, 2008, 71.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, 168.

[Col+11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. "Natural language processing (almost) from scratch". In: *Journal of machine learning research* 12.Aug (2011), 2493.

[CS04] Christoph Csallner and Yannis Smaragdakis. "JCrasher: an automatic robustness tester for Java". In: *Software Practice and Experience* 34.11 (2004), 1025.

[Dal+06] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. "Mining object behavior with ADABU". In: *Workshop on Dynamic Systems Analysis (WODA)*. ACM, 2006, 17.

[DTR18] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. "Path-based function embedding and its application to error-handling specification mining". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 2018, 423.

[Ede+02] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. "Multithreaded Java program test generation". In: *IBM Systems Journal* 41.1 (2002), 111.

[Ern+16] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. "Locking discipline inference and checking". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, 1133.

[EZG15] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. "Web API growing pains: Loosely coupled yet strongly tied". In: *Journal of Systems and Software (JSS)* 100 (2015), 27.

[Fen+03] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong. "Anomaly detection using call stack information". In: *2003 Symposium on Security and Privacy, 2003*. 2003, 62.

[FN99] Norman E Fenton and Martin Neil. "A critique of software defect prediction models". In: *IEEE Transactions on software engineering* 25.5 (1999), 675.

[FF04] Cormac Flanagan and Stephen N. Freund. "Atomizer: a dynamic atomicity checker for multithreaded programs". In: *Symposium on Principles of Programming Languages (POPL)*. ACM, 2004, 256.

[FF09] Cormac Flanagan and Stephen N. Freund. "FastTrack: efficient and precise dynamic race detection". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009, 121.

[FF00] Cormac Flanagan and Stephen N. Freund. "Type-based race detection for Java". In: 2000, 219.

[FQ03] Cormac Flanagan and Shaz Qadeer. "A type and effect system for atomicity". In: ACM, 2003, 338.

[Fra+05] Eibe Frank, Mark A. Hall, Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. "WEKA - A Machine Learning Workbench for Data Mining". In: *The Data Mining and Knowledge Discovery Handbook*. 2005, 1305.

[FA11] Gordon Fraser and Andrea Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". In: *SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 2011, 416.

[GS08] Mark Gabel and Zhendong Su. "Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces". In: *Symposium on Foundations of Software Engineering (FSE)*. ACM, 2008, 339.

[GZ13] Francis Galiegue and Kris Zyp. *JSON Schema draft 04*. 2013.

[GRS04] Debin Gao, Michael K. Reiter, and Dawn Song. "Gray-Box Extraction of Execution Graphs for Anomaly Detection". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: Association for Computing Machinery, 2004, 318–329.

[GFW03] Thomas Gärtner, Peter Flach, and Stefan Wrobel. "On graph kernels: Hardness results and efficient alternatives". In: *Learning Theory and Kernel Machines*. Springer, 2003, 129.

[Gas+13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Structural Detection of Android Malware Using Embedded Call Graphs". In: *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*. AISec '13. Berlin, Germany: ACM, 2013, 45.

[GSC99] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM". In: (1999).

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, 213.

[Gof+16] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. "Automatic generation of oracles for exceptional behaviors". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016, 213.

[GFZ12] Florian Gross, Gordon Fraser, and Andreas Zeller. "Search-Based System Testing: High Coverage, No False Alarms". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2012, 67.

[GWZ10] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. "Learning from 6,000 projects: Lightweight cross-project anomaly detection". In: *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2010, 119.

[GZK18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search". In: *ICSE*. 2018.

[Gup+17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *AAAI*. 2017.

[HP18a] Andrew Habib and Michael Pradel. "How Many of All Bugs Do We Find? A Study of Static Bug Detectors". In: *ASE*. 2018.

[HP18b] Andrew Habib and Michael Pradel. "Is This Class Thread-Safe? Inferring Documentation using Graph-based Learning". In: *ASE*. 2018.

[HP19] Andrew Habib and Michael Pradel. "Neural Bug Finding: A Study of Opportunities and Challenges". In: *CoRR* abs/1906.00307 (2019).

[Hab+19] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. "Type Safety with JSON Subschema". In: *CoRR* abs/1911.12651 (2019).

[Hag19] Petter Haggholm. *is-json-schema-subset*. 2019. URL: https://github.com/haggholm/is-json-schema-subset.

[HL02] Sudheendra Hangal and Monica S. Lam. "Tracking down software bugs using automatic anomaly detection". In: *International Conference on Software Engineering (ICSE)*. ACM, 2002, 291.

[HB07] Zaïd Harchaoui and Francis Bach. "Image classification with segmentation graph kernels". In: *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*. IEEE. 2007, 1.

[Har+18] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Sang Peter Chin. "Learning to Repair Software Vulnerabilities with Generative Adversarial Networks". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.* 2018, 7944.

[Hel+18] V. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis. "Deep Learning Type Inference". In: *FSE*. 2018.

[HRD08] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. "Developing and debugging algebraic specifications for Java classes". In: *ACM Transactions on Software Engineering and Methodology* 17.3 (2008), 1.

[HRD07] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. "Discovering Documentation for Java Container Classes". In: *IEEE Transactions on Software Engineering* 33.8 (2007), 526.

[Hin+12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. "On the naturalness of software". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 2012, 837.

[Hir+19] Martin Hirzel, Kiran Kate, Avraham Shinnar, Subhrajit Roy, and Parikshit Ram. *Type-Driven Automated Learning with Lale*. 2019.

[Hoa+20] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. "CC2Vec: Distributed Representations of Code Changes". In: *ICSE*. 2020.

[HP03] Haruo Hosoya and Benjamin C. Pierce. "XDuce: A Statically Typed XML Processing Language". In: *Transactions on Internet Technology (TOIT)* 3.2 (2003), 117.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. "Regular Expression Types for XML". In: *Transactions on Programming Languages and Systems (TOPLAS)* 27.1 (2005), 46.

[HP04] David Hovemeyer and William Pugh. "Finding bugs is easy". In: *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2004, 132.

[Snoa] *Introducing SchemaVer for semantic versioning of schemas*. URL: https://snowplowanalytics.com/blog/2014/05/13/introducing-schemaver-for-semantic-versioning-of-schemas/.

[Tsba] *JBoss Platform issue 1416472*. URL: https://bugzilla.redhat.com/show_bug.cgi?id=1416472.

[Jia+17] He Jiang, Jingxuan Zhang, Zhilei Ren, and Tao Zhang. "An Unsupervised Approach for Discovering Relevant Tutorial Fragments for APIs". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, 38.

[Joh+13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, 672.

[Joh78] S. C. Johnson. *Lint, a C Program Checker*. Murray Hill: Bell Telephone Laboratories, 1978.

[Jos+09] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. "CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs". In: *Conference on Computer Aided Verification*. Springer, 2009, 675.

[Jsoa] *JSON Schema Compare*. 2017. URL: https://github.com/mokkabonna/json-schema-compare.

[Jsob] *JSON Schema Diff Validator*. 2017. URL: https://bitbucket.org/atlassian/json-schema-diff-validator.

[Jsoc] *JSON Schema Test Suite*. 2012. URL: https://github.com/json-schema-org/JSON-Schema-Test-Suite.

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: a database of existing faults to enable controlled testing studies for Java programs". In: *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 2014, 437.

[Jus+14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. "Are mutants a valid substitute for real faults in software testing?" In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, 654.

[Kam+07] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. "The effects of over and under sampling on fault-prone module detection". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE. 2007, 196.

[KTI03] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. "Marginalized Kernels Between Labeled Graphs". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. Washington, DC, USA: AAAI Press, 2003, 321.

[KFY18] Jinhan Kim, Robert Feldt, and Shin Yoo. "Guiding Deep Learning System Testing using Surprise Adequacy". In: *CoRR* abs/1808.08444 (2018).

[KE07] Sunghun Kim and Michael D. Ernst. "Which warnings should I fix first?" In: *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, 45.

[Kin76] J. C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (1976), 385.

[KL02] Risi Imre Kondor and John D. Lafferty. "Diffusion Kernels on Graphs and Other Discrete Input Spaces". In: *Proceedings of the Nineteenth International Conference on Machine Learning*. ICML '02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, 315.

[KE03] Ted Kremenek and Dawson R. Engler. "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations". In: *International Symposium on Static Analysis (SAS)*. Springer, 2003, 295.

[Kuba] *Kubernetes JSON Schemas*. 2019. URL: https://github.com/instrumenta/kubernetes-json-schema.

[Lak75] George Lakoff. "Hedges: A study in meaning criteria and the logic of fuzzy concepts". In: *Contemporary research in philosophical logic and linguistic semantics*. Springer, 1975, 221.

[Lal] *Lale: Python library for semi-automated data science*. 2019. URL: https://github.com/ibm/lale.

[Laz] *Lazy initialization*. URL: http://www.javapractices.com/topic/TopicAction.do?Id=34.

[LCR11] Choonghwan Lee, Feng Chen, and Grigore Rosu. "Mining Parametric Specifications". In: *International Conference on Software Engineering (ICSE)*. 2011, 591.

[Leg+16] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Rosu, and Darko Marinov. "How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 2016, 602.

[LSF03] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. "How Software Engineers Use Documentation: The State of the Practice". In: *IEEE Software* 20.6 (2003), 35.

[LW90] Hareton K. N. Leung and Lee White. "Insights into testing and regression testing global variables". In: *Journal of Software Maintenance* 2.4 (1990), 209.

[Li+13] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. "How Does Web Service API Evolution Affect Clients?" In: *International Conference on Web Services (ICWS)*. 2013, 300.

[Li+19] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. "Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks". In: *OOPSLA*. 2019.

[Li+18] Zhen Li, Shouhuai Xu Deqing Zou and, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *NDSS*. 2018.

[LZ05] Zhenmin Li and Yuanyuan Zhou. "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code". In: *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, 306.

[Lin+16] Z. Lin, H. Zhong, Y. Chen, and J. Zhao. "LockPeeker: Detecting latent locks in Java APIs". In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, 368.

[Lio96] J. L. Lions. *ARIANE 5 Flight 501 Failure. Report by the Inquiry Board*. European Space Agency. 1996.

[Liu+19] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. "Learning to spot and refactor inconsistent method names". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, 1.

[Lu+06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. "AVIO: detecting atomicity violations via access interleaving invariants". In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, 37.

[Lu+05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. "Bugbench: Benchmarks for evaluating bug detection tools". In: *Workshop on the Evaluation of Software Defect Detection Tools*. 2005.

[Lu+08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008, 329.

[LC09] Brandon Lucia and Luis Ceze. "Finding concurrency bugs with context-aware communication graphs". In: *Symposium on Microarchitecture (MICRO)*. ACM, 2009, 553.

[MPP19] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. "NL2Type: inferring JavaScript function types from natural language information". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, 304.

[Mar+17] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset". In: *Empirical Software Engineering* 22.4 (2017), 1936.

[McB+17] P. W. McBurney, S. Jiang, M. Kessentini, N. A. Kraft, A. Armaly, M. W. Mkaouer, and C. McMillan. "Towards Prioritizing Documentation Effort". In: *IEEE Transactions on Software Engineering* PP.99 (2017), 1.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.

[McK98] William M. McKeeman. "Differential Testing for Software". In: *Digital Technical Journal* 10.1 (1998), 100.

[MS96] Maged M Michael and Michael L Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM. 1996, 267.

[Mik+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.* 2013, 3111.

[MBM10] Martin Monperrus, Marcel Bruch, and Mira Mezini. "Detecting Missing Method Calls in Object-Oriented Software". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2010, 2.

[Mon+12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. "What should developers be aware of? An empirical study on the directives of API documentation". In: *Empir. Softw. Eng.* 17.6 (2012), 703.

[MB19] Manish Motwani and Yuriy Brun. "Automatically generating precise Oracles from structured natural language specifications". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, 188.

[Mou+16] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. "Convolutional Neural Networks over Tree Structures for Programming Language Processing". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. 2016, 1287.

[MCJ17] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. "Bayesian Specification Learning for Finding API Usage Errors". In: *FSE*. 2017.

[Mus+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. "Finding and Reproducing Heisenbugs in Concurrent Programs". In: *Symposium on Operating Systems Design and Implementation*. USENIX, 2008, 267.

[Nai+09] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. "Effective static deadlock detection". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2009, 386.

[New15] Sam Newman. *Building Microservices: Designing Fine Grained Systems*. O'Reilly, 2015.

[Ngu+19] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, 819.

[Ngu+09] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. "Graph-based mining of multiple object usage patterns". In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, 383.

[Nis+12] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. "BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code". In: *International Conference on Software Engineering (ICSE)*. 2012, 727.

[Tsbb] *NX issue 239*. URL: https://track.radensolutions.com/issue/NX-239.

[OC03] Robert O'Callahan and Jong-Deok Choi. "Hybrid dynamic data race detection". In: *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 2003, 167.

[Oca+13] Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. "An Empirical Study of Client-Side JavaScript Bugs". In: *Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2013, 55.

[Pac+07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2007, 75.

[Pal+11] Nicolas Palix, Gaël Thomas 0001, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. "Faults in linux: ten years later". In: *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011, 305.

[PKJ09] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14.3 (2009), 286.

[Pan+12] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit M. Paradkar. "Inferring method specifications from natural language API descriptions". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, 815.

[PSW76] David L. Parnas, John E. Shore, and David Weiss. "Abstract types defined as classes of variables". In: *Conference on Data: Abstraction, Definition and Structure*. 1976, 149.

[Pea+17] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. "Evaluating and improving fault localization". In: *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*. IEEE. 2017, 609.

[Pez+16] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. "Foundations of JSON Schema". In: *International Conference on World Wide Web (WWW)*. 2016, 263.

[Pha+17] H. Phan, H. A. Nguyen, T. N. Nguyen, and H. Rajan. "Statistical Learning for Inference between Implementations and Documentation". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. 2017, 27.

[Pie+15] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas J. Guibas. "Learning Program Embeddings to Propagate Feedback on Student Code". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, 1093.

[PH13] Jacques A. Pienaar and Robert Hundt. "JSWhiz: Static analysis for JavaScript memory leaks". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. 2013, 11:1.

[PKN16] Martin Pilat, Tomas Kren, and Roman Neruda. "Asynchronous Evolution of Data Mining Workflow Schemes by Strongly Typed Genetic Programming". In: *International Conference on Tools with Artificial Intelligence (ICTAI)*. 2016, 577.

[Pou04] Kevin Poulsen. *Software Bug Contributed to Blackout*. SecurityFocus. 2004.

[PG09] Michael Pradel and Thomas R. Gross. "Automatic Generation of Object Usage Specifications from Large Method Traces". In: *International Conference on Automated Software Engineering (ASE)*. 2009, 371.

[PG13] Michael Pradel and Thomas R. Gross. "Automatic Testing of Sequential and Concurrent Substitutability". In: *International Conference on Software Engineering (ICSE)*. 2013, 282.

[PG11] Michael Pradel and Thomas R. Gross. "Detecting anomalies in the order of equally-typed method arguments". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2011, 232.

[PG12] Michael Pradel and Thomas R. Gross. "Fully Automatic and Precise Detection of Thread Safety Violations". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2012, 521.

[PHG14] Michael Pradel, Markus Huggler, and Thomas R. Gross. "Performance Regression Testing of Concurrent Classes". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2014, 13.

[PS18] Michael Pradel and Koushik Sen. "DeepBugs: A learning approach to name-based bug detection". In: *PACMPL* 2.OOPSLA (2018), 147:1.

[Pra+14] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. "EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2014, 33.

[Pra+12] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. "Statically Checking API Protocol Conformance with Mined Multi-Object Specifications". In: *International Conference on Software Engineering (ICSE)*. 2012, 925.

[Pra+20] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. "TypeWriter: Neural Type Prediction with Search-based Validation". In: ESEC/SIGSOFT FSE. 2020.

[PG01] Christoph von Praun and Thomas R. Gross. "Object Race Detection". In: *Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2001, 70.

[PG03] Christoph von Praun and Thomas R. Gross. "Static conflict analysis for multi-threaded object-oriented programs". In: *Conference on Programming Languages Design and Implementation*. ACM, 2003, 115.

[Kubb] *Production-Grade Container Orchestration*. URL: https://kubernetes.io/.

[Pro] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers.

[Rah+14] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. "Comparing static bug finders and statistical prediction". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, 424.

[Ral+05] Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, and Pierre Baldi. "Graph kernels for chemical informatics". In: *Neural networks* 18.8 (2005), 1093.

[RG03] Jan Ramon and Thomas Gärtner. "Expressivity versus efficiency of graph kernels". In: *Proceedings of the first international workshop on mining graphs, trees and sequences*. 2003, 65.

[RR17] Inderjot Kaur Ratol and Martin P. Robillard. "Detecting fragile comments". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, 112.

[Ray+16] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. "On the "naturalness" of buggy code". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, 428.

[RVK15] Veselin Raychev, Martin T. Vechev, and Andreas Krause. "Predicting Program Properties from "Big Code"." In: *Principles of Programming Languages (POPL)*. 2015, 111.

[RVY14] Veselin Raychev, Martin T. Vechev, and Eran Yahav. "Code completion with statistical language models". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 2014, 44.

[RW12] Eric Redmond and Jim R. Wilson. *Seven Databases in Seven Weeks*. The Pragmatic Bookshelf, 2012.

[Tsbc] *RESTEasy issue 1669*. URL: https://issues.jboss.org/browse/RESTEASY-1669.

[Ric+17] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. "Detecting Argument Selection Defects". In: *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2017.

[Rob09] Martin P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE Software* 26.6 (2009), 27.

[Rod+16] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices". In: *International Conference on Web Engineering (ICWE)*. 2016, 21.

[Rot+01] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. "Prioritizing test cases for regression testing". In: *IEEE Transactions on Software Engineering* 27.10 (2001), 929.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. "A Comparison of Bug Finding Tools for Java". In: *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 2004, 245.

[Rut+08] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. "Predicting accurate and actionable static analysis warnings: an experimental approach". In: *International Conference on Software Engineering (ICSE)*. 2008, 341.

[Sac+18] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. "Retrieval on source code: a neural code search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM. 2018, 31.

[Sad+15] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. "Tricorder: Building a Program Analysis Ecosystem". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, 598.

[SR14] Malavika Samak and Murali Krishna Ramanathan. "Multithreaded Test Synthesis for Deadlock Detection". In: *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 2014, 473.

[SR15] Malavika Samak and Murali Krishna Ramanathan. "Synthesizing tests for detecting atomicity violations". In: *ESEC/FSE*. 2015, 131.

[SRJ15] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. "Synthesizing racy tests." In: *PLDI*. 2015, 175.

[STR16] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. "Directed synthesis of failing concurrent executions". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 2016, 430.

[Sav+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. "Eraser: A Dynamic Data Race Detector for Multithreaded Programs". In: *ACM Transactions on Computer Systems* 15.4 (1997), 391.

[Sca+09] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The graph neural network model". In: *IEEE Transactions on Neural Networks* 20.1 (2009), 61.

[Sch+18] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. "Automating Large-scale Data Quality Verification". In: *Conference on Very Large Data Bases (VLDB)*. 2018, 1781.

[SS02] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. the MIT Press, 2002.

[Seg+16] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. "A survey on metamorphic testing". In: *IEEE Transactions on software engineering* 42.9 (2016), 805.

[Sek+01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. "A fast automaton-based method for detecting anomalous program behaviors". In: *Symposium on Security and Privacy (SSP)*. IEEE, 2001, 144.

[SP16] Marija Selakovic and Michael Pradel. "Performance Issues and Optimizations in JavaScript: An Empirical Study". In: *International Conference on Software Engineering (ICSE)*. 2016, 61.

[Sem] *Semantic Versioning 2.0.0*. URL: https://semver.org/.

[Seno8] Koushik Sen. "Race directed random testing of concurrent programs". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2008, 11.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005, 263.

[Sen+13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript". In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2013, 488.

[Sha+15] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 2015, 201.

[She+11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. "Weisfeiler-Lehman Graph Kernels". In: *J. Mach. Learn. Res.* 12 (2011), 2539.

[Sho+07] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. "Static specification mining using automata-based abstractions". In: *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2007, 174.

[Smi+19] Micah J. Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. *The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development*. 2019.

[Sno14] Snowplow Analytics. *Central repository for storing JSON Schemas, Avros and Thrifts*. 2014. URL: https://github.com/snowplow/iglu-central.

[Son+11] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. "A general software defect-proneness prediction framework". In: *IEEE transactions on software engineering* 37.3 (2011), 356.

[Sun+07] Yanmin Sun, Mohamed S Kamel, Andrew KC Wong, and Yang Wang. "Cost-sensitive boosting for classification of imbalanced data". In: *Pattern Recognition* 40.12 (2007), 3358.

[SSZ12] Zhongbin Sun, Qinbao Song, and Xiaoyan Zhu. "Using coding-based ensemble learning to improve software defect prediction". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), 1806.

[Swaa] *Swagger::Diff*. 2015. URL: https://github.com/civisanalytics/swagger-diff.

[Swab] *Swagger/OpenAPI Specification*. URL: https://swagger.io/.

[Swa+05] S Joshua Swamidass, Jonathan Chen, Jocelyne Bruand, Peter Phung, Liva Ralaivola, and Pierre Baldi. "Kernels for small molecules and the prediction of mutagenicity, toxicity and anti-cancer activity". In: *Bioinformatics* 21.suppl 1 (2005), i359.

[TZP11] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. "aComment: mining annotations from comments and code to detect interrupt related concurrency bugs". In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ACM, 2011, 11.

[Tan+07] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. "/*icomment: bugs or bad comments?*/". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by Thomas C. Bressoud and M. Frans Kaashoek. ACM, 2007, 145.

[Tan+12] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. "@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. Ed. by Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche. IEEE Computer Society, 2012, 260.

[Tem+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. "Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies". In: *Asia Pacific Software Engineering Conference (APSEC)*. 2010.

[TC16] Valerio Terragni and Shing-Chi Cheung. "Coverage-Driven Test Code Generation for Concurrent Classes". In: *ICSE*. 2016.

[Dou] *The "Double-Checked Locking is Broken" Declaration*. URL: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[Snob] *The enterprise-grade event data collection platform*. URL: https://snowplowanalytics.com/.

[Wp2] *The Washington Post ANS specification*. 2015. URL: https://github.com/washingtonpost/ans-schema.

[TX09] Suresh Thummalapenta and Tao Xie. "Mining Exception-Handling Rules as Sequence Association Rules". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2009, 496.

[Thu+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. "Synthesizing method sequences for high-coverage testing." In: *OOPSLA*. 2011, 189.

[Thu+12] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools". In: *Conference on Automated Software Engineering (ASE)*. ACM, 2012, 50.

[Thu+15] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. "To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools". In: *Autom. Softw. Eng.* 22.4 (2015), 561.

[Tia+17] Yuan Tian, Ferdian Thung, Abhishek Sharma, and David Lo. "APIBot: question answering bot for API documentation". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 2017, 153.

[Tom+19] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. "BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes". In: *Proceedings of the 41st International Conference*

on Software Engineering. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, 339–349.

[TH03] Akihiko Tozawa and Masami Hagiya. "XML Schema Containment Checking based on Semi-implicit Techniques". In: *International Conference on Implementation and Application of Automata (CIAA)*. 2003, 213.

[TR16] Christoph Treude and Martin P. Robillard. "Augmenting API Documentation with Insights from Stack Overflow". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, 392.

[TRA20] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. "Learning to Encode and Classify Test Executions". In: *CoRR* (2020).

[Tuf+19] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. "On learning meaningful code changes via neural machine translation". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 2019, 25.

[VR+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. "Soot - a Java bytecode optimization framework". In: *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM, 1999, 125.

[VLK02] Meenakshi Vanmali, Mark Last, and Abraham Kandel. "Using a neural network in the software testing process". In: *Int. J. Intell. Syst.* 17.1 (2002), 45.

[Vas+19] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. "Neural Program Repair by Jointly Learning to Localize and Repair". In: *ICLR*. 2019.

[VCD17] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. "Recovering clear, natural identifiers from obfuscated JS names". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 2017, 683.

[Vis+10] S. V. N. Vishwanathan, Nicol N. Schraudolph, Risi Kondor, and Karsten M. Borgwardt. "Graph Kernels". In: *J. Mach. Learn. Res.* 11 (2010), 1201.

[Vis+03] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. "Model Checking Programs". In: *Automated Software Engineering* 10.2 (2003), 203.

[Vis+14] Bimal Viswanath, M. Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. "Towards Detecting Anomalous User Behavior in Online Social Networks." In: *USENIX Security*. 2014, 223.

[Vit+14] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. "Design and Evaluation of Gradual Typing for Python". In: *Dynamic Languages Symposium (DLS)*. 2014, 45.

[Wag+09] C. Wagner, G. Wagener, R. State, and T. Engel. "Malware analysis with graph kernels and support vector machines". In: *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*. 2009, 63.

[Wag+05] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. "Comparing Bug Finding Tools with Reviews and Tests". In: *International Conference on Testing of Communicating Systems (TestCom)*. Springer, 2005, 40.

[WSS17] Ke Wang, Rishabh Singh, and Zhendong Su. "Dynamic Neural Program Embedding for Program Repair". In: *CoRR* abs/1711.07163 (2017).

[WS20] Ke Wang and Zhendong Su. "Blended, Precise Semantic Program Embeddings". In: PLDI 2020. London, UK: Association for Computing Machinery, 2020, 121–134.

[WS06] Liqiang Wang and Scott D. Stoller. "Accurate and efficient runtime detection of atomicity errors in concurrent programs". In: *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*. ACM, 2006, 137.

[WY13] Shuo Wang and Xin Yao. "Using class imbalance learning for software defect prediction". In: *IEEE Transactions on Reliability* 62.2 (2013), 434.

[WLT16] Song Wang, Taiyue Liu, and Lin Tan. "Automatically learning semantic features for defect prediction". In: *ICSE*. 2016, 297.

[Wan+16] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. "Bugram: bug detection with n-gram language models". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 2016, 708.

[Wan+19] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. "Learning a Static Bug Finder from Data". In: *CoRR* abs/1907.05579 (2019).

[WM03] Takashi Washio and Hiroshi Motoda. "State of the Art of Graph-based Data Mining". In: *SIGKDD Explor. Newsl.* 5.1 (2003), 59.

[WZ09] Andrzej Wasylkowski and Andreas Zeller. "Mining Temporal Specifications from Object Usage". In: *International Conference on Automated Software Engineering (ASE)*. IEEE, 2009, 295.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. "Detecting object usage anomalies". In: *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, 35.

[Wat+20] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. "On Learning Meaningful Assert Statements for Unit Test Cases". In: *ICSE*. 2020.

[WL68] Boris Weisfeiler and AA Lehman. "A reduction of a graph to a canonical form and an algebra arising during this reduction". In: *Nauchno-Technicheskaya Informatsia* 2.9 (1968), 12.

[WCB14] Jason Weston, Sumit Chopra, and Antoine Bordes. "Memory Networks". In: *CoRR* abs/1410.3916 (2014).

[WML02] John Whaley, Michael C. Martin, and Monica S. Lam. "Automatic Extraction of Object-Oriented Component Interfaces". In: *Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2002, 218.

[Whi+16] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. "Deep learning code fragments for code clone detection". In: *ASE*. 2016, 87.

[WTE05] Amy Williams, William Thies, and Michael D. Ernst. "Static Deadlock Detection for Java Libraries". In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2005, 602.

[WO04] Henry Wong and Scott Oaks. *Java Threads*. 3rd edition. O'Reilly Media, Inc., 2004.

[Xia+12] Xusheng Xiao, Amit M. Paradkar, Suresh Thummalapenta, and Tao Xie. "Automated extraction of security policies from natural-language software documents". In: *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. Ed. by Will Tracz, Martin P. Robillard, and Tevfik Bultan. ACM, 2012, 12.

[Xie+05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution". In: *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2005, 365.

[XBH05] Min Xu, Rastislav Bodík, and Mark D. Hill. "A serializability violation detector for shared-memory server programs". In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, 1.

[Yan+06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. "Perracotta: Mining temporal API rules from imperfect traces". In: *International Conference on Software Engineering (ICSE)*. ACM, 2006, 282.

[Yin+18] Pengcheng Yin, Graham Neubig, Marc Brockschmidt Miltiadis Allamanis and, and Alexander L. Gaunt. "Learning to Represent Edits". In: *CoRR* 1810.13337 (2018).

[Zha+20] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. "CPC: Automatically Classifying and Propagating Natural Language Comments via Program Analysis". In: *ICSE*. 2020.

[Zha+19] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. "A Novel Neural Source Code Representation based on Abstract Syntax Tree". In: *ICSE*. 2019.

[Zha+14] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, 1105.

[Zha18] Zijun Zhang. "Improved Adam optimizer for deep neural networks". In: *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*. IEEE. 2018, 1.

[ZH18] Gang Zhao and Jeff Huang. "DeepSim: deep learning code functional similarity". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 2018, 141.

[Zha+18] Rui Zhao, David Bieber, Kevin Swersky, and Daniel Tarlow. "Neural Networks for Modeling Source Code Edits". In: (2018).

[Zhe+06] Jiang Zheng, Laurie A. Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. "On the Value of Static Analysis for Fault Detection in Software". In: *IEEE Trans. Software Eng.* 32.4 (2006), 240.

[ZC09] Michael Zhivich and Robert K. Cunningham. "The Real Cost of Software Errors". In: *IEEE Security & Privacy* 7.2 (2009), 87.

[ZS13] Hao Zhong and Zhendong Su. "Detecting API Documentation Errors". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, 803–816.

[ZZM08] Hao Zhong, Lu Zhang, and Hong Mei. "Inferring Specifications of Object Oriented APIs from API Source Code". In: *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2008, 221.

[Zho+09a] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. "Inferring Resource Specifications from Natural Language API Documentation". In: *International Conference on Automated Software Engineering (ASE)*. 2009, 307.

[Zho+09b] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. "MAPO: Mining and Recommending API Usage Patterns". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2009, 318.

[Zho+17] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. "Analyzing APIs Documentation and Code to Detect Directive Defects". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, 27.

[Zim+09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process". In: *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, 91.

[Zyp09] Kris Zyp. *JSON Schema*. 2009.